

Sistemi Operativi Teoria

Parte 2 – Gestione dei processi

Appunti a cura di Liccardo Giuseppe
Università degli Studi di Napoli “Parthenope”



Indice – Parte 2

CAPITOLO 5. Processi e thread	4
5.1 Processi e programmi	4
5.1.1 <i>Cos'è un processo?</i>	4
5.1.2 <i>Relazioni tra processi e programmi</i>	4
5.1.3 <i>Processi figli</i>	5
5.1.4 <i>Concorrenza e parallelismo</i>	5
5.2 Implementazione dei processi.....	5
5.2.1 <i>Stati del processo e transizioni di stato</i>	6
5.2.2 <i>Il contesto del processo e il process control block</i>	8
5.2.3 <i>Salvataggio del contesto, scheduling e dispatching</i>	8
5.2.4 <i>Gestione degli eventi</i>	10
5.2.5 <i>Condivisione, comunicazione e sincronizzazione tra processi</i>	11
5.2.6 <i>Segnali</i>	11
5.3 Thread.....	11
5.3.1 <i>Thread di livello kernel, di livello utente e ibridi</i>	13
5.4 Casi di studio.....	14
5.4.1 <i>Processi in Unix</i>	14
CAPITOLO 6. La gestione della CPU – Scheduling	16
6.1 Scheduling: terminologia e concetti	16
6.1.1 <i>Indicatori di prestazioni</i>	16
6.1.2 <i>Tecniche fondamentali di scheduling</i>	18
6.1.3 <i>Il ruolo della priorità</i>	18
6.2 Classificazione delle attività di scheduling.....	18
6.2.1 <i>Gestione degli eventi e schedulazione</i>	19
6.2.2 <i>Scheduler a lungo, medio e breve termine</i>	19
6.2.3 <i>Comportamento dei processi (cicli di CPU, cicli di I/O)</i>	20
6.2.4 <i>Politiche di scheduling della CPU</i>	20
6.2.5 <i>Algoritmi di schedulazione</i>	21
6.3 Politiche di scheduling nonpreemptive	21
6.3.1 <i>Scheduling First Come First Served (FCFS)</i>	22
6.3.2 <i>Scheduling Shortest Job Firts (SJF) – Shortest Next Process First (SNPF)</i>	23
6.3.3 <i>Scheduling Highest Response Ratio Next (HRRN)</i>	24
6.4 Politiche di scheduling preemptive	25
6.4.1 <i>Scheduling Shortest Job Firts (SJF) – Shortest Remaining Time First (SRTF)</i>	25
6.4.2 <i>Scheduling Round-Robin con Time-Slicing (RR)</i>	26
6.4.3 <i>Scheduling Least Completed Next (LCN)</i>	27
6.5 Scheduling in pratica	29
6.5.1 <i>Meccanismi di scheduling</i>	29
6.5.2 <i>Scheduling basato su priorità</i>	29

6.5.3 <i>Scheduling multilivello</i>	29
6.5.4 <i>Scheduling multilivello adattivo</i>	30
6.5.5 <i>Scheduling fair share</i>	30
6.6 <i>Scheduling real-time</i>	30
6.6.1 <i>Precedenze e schedulabilità di un processo</i>	31
6.6.2 <i>Scheduling della deadline</i>	31
6.7 <i>Casi di studio</i>	32
6.7.1 <i>Scheduling in Unix</i>	32
6.7.2 <i>Scheduling in Linux</i>	32
CAPITOLO 7. Sincronizzazione dei processi: memoria condivisa.....	33
7.1 Cos'è la sincronizzazione dei processi?	33
7.1.1 <i>Processi concorrenti</i>	33
7.1.2 <i>Interazione tra processi</i>	33
7.2 <i>Race condition</i>	35
7.2.1 <i>Condizioni di Bernstein</i>	35
7.3 Sincronizzazione per l'accesso ai dati e sezioni critiche	34
7.3.1 <i>Proprietà dell'implementazione di una sezione critica</i>	34
7.4 Sincronizzazione per il controllo e operazioni indivisibili	34
7.5 Approcci alla sincronizzazione	35
7.5.1 <i>Ciclare – Approccio software</i>	35
7.5.2 <i>Bloccare – Approccio hardware</i>	36
7.6 Approccio hardware	36
7.6.1 <i>Disabilitazione delle interruzioni</i>	36
7.6.2 <i>Proprietà dell'approccio hardware</i>	37
7.7 Approccio software	38
7.7.1 <i>Algoritmi per due processi</i>	38
7.7.2 <i>Algoritmi per n processi</i>	42
7.8 Approccio SO/Linguaggio – Semafori	43
7.8.1 <i>Semafori</i>	43
7.8.2 <i>Uso dei semafori nei sistemi concorrenti</i>	44
7.9 Approccio SO/Linguaggio – Monitor	46
7.9.1 <i>Struttura di un monitor</i>	46
7.9.2 <i>Variabili di condizione</i>	46
7.9.3 <i>Semafori vs. monitor</i>	47
7.10 Problemi classici di sincronizzazione dei processi	47
7.11 Alcune soluzioni.....	49
7.11.1 <i>Produttore-consumatore mediante sezioni critiche</i>	49
7.11.2 <i>Produttore-consumatore mediante semafori</i>	50
7.11.3 <i>Produttore-consumatore mediante monitor</i>	51
7.11.4 <i>Lettore-scrittore con priorità ai lettori</i>	52
7.11.5 <i>Lettore-scrittore con priorità agli scrittori</i>	53
7.11.6 <i>Filosofi a cena</i>	55

7.11.7 <i>Filosofi a cena con monitor</i>	56
7.12 Casi di studio: sincronizzazione dei thread POSIX	58
CAPITOLO 8. Sincronizzazione dei processi: message passing	59
8.1 Panoramica sul message passing.....	59
8.1.1 <i>Send bloccanti e non-bloccanti</i>	59
8.1.2 <i>Message passing sincrono e asincrono</i>	59
8.1.3 <i>Denominazione diretta e indiretta</i>	60
8.2 Produttore-consumatore mediante scambio di messaggi	61
8.3 Casi di studio.....	61
8.3.1 <i>Message passing in Unix</i>	61
CAPITOLO 9. Deadlock.....	64
9.1 Definizione di deadlock	64
9.2 I deadlock nell'allocazione delle risorse	64
9.2.1 <i>Condizioni per un deadlock di risorsa</i>	65
9.2.2 <i>Modelli dello stato di allocazione delle risorse</i>	65
9.3 Gestione dei deadlock	66
9.4 Individuazione e risoluzione dei deadlock.....	67
9.4.1 <i>Individuazione dei deadlock</i>	67
9.4.2 <i>Risoluzione dei deadlock</i>	68
9.5 Caratterizzazione dei deadlock delle risorse tramite i modelli a grafi.....	68
9.5.1 <i>Sistemi Singola-Istanza</i>	68
9.5.2 <i>Sistemi Istanza-Multipla</i>	68
9.6 Prevenzione dei deadlock.....	69
9.7 Evitare i deadlock	70
9.7.1 <i>Algoritmo del banchiere</i>	70
9.8 Casi di studio.....	73
9.9.1 <i>Gestione dei deadlock in Unix</i>	73

CAPITOLO 5. Processi e thread

Un **processo** è un programma in esecuzione che utilizza un insieme di risorse. Un'applicazione può essere sviluppata per avere molti processi che operano concorrentemente e interagiscono tra di loro per ottenere un risultato comune. In questo modo, l'applicazione può essere in grado di fornire una risposta più veloce all'utente. Un SO mantiene in esecuzione un gran numero di processi per ogni istante di tempo. E' il kernel che alloca le risorse ai processi e li schedula per l'utilizzo della CPU.

Gestire i processi significa: crearli, soddisfare le richieste di risorse, schedularli per l'uso della CPU, sincronizzarli per controllare la loro interazione, evitare deadlock in modo che non siano in attesa l'un l'altro indefinitamente e terminarli quando hanno concluso l'esecuzione.

Anche un **thread** è un programma in esecuzione ma usa le risorse di un processo, quindi somiglia a un processo in tutti gli aspetti. La gestione dei thread genera meno overhead rispetto alla gestione dei processi. E' possibile che molti thread vengano eseguiti nell'ambito dello stesso processo.

Spiegheremo, in questo capitolo, come il kernel controlla i processi e i thread, come tiene traccia dei loro *stati* e come usa le informazioni di stato per organizzare la loro esecuzione. Discuteremo inoltre di come un programma può creare processi o thread concorrenti e di come questi possono interagire per raggiungere un obiettivo comune.

5.1 Processi e programmi

Un programma è un'entità passiva che non effettua nessuna operazione da solo, mentre un processo è un programma in esecuzione, quindi esegue le azioni specificate all'interno del programma.

5.1.1 Cos'è un processo?

Un **processo** è un programma in esecuzione che utilizza le risorse a esso allocate. Solitamente, un'applicazione è costituita da più processi che interagiscono tra loro per il raggiungimento di un obiettivo comune.

Formalmente, ognuno di questi processi è costituito da due parti: una parte *statica*, ovvero il codice; una parte *dinamica*, ovvero le risorse che utilizza. Il codice del programma è costante, mentre la parte dinamica varia nel tempo.

Nello specifico, un processo comprende sei componenti:

- *id* – identificativo univoco assegnato dal SO
- *codice* – codice del programma
- *dati* – dati usati durante l'esecuzione del programma, inclusi i dati contenuti nei file
- *stack* – contiene i parametri e gli indirizzi di ritorno delle funzioni chiamate durante l'esecuzione
- *risorse* – risorse allocate dal SO
- *stato della CPU* – composto dal contenuto del PSW e dei registri GPR della CPU

Lo stato della CPU contiene una serie di informazioni molto importanti come la prossima istruzione da eseguire e il contenuto del campo CC (condition code). Inoltre lo stato della CPU cambia man mano che l'esecuzione del programma progredisce.

5.1.2 Relazioni tra processi e programmi

Un programma consiste di un insieme di funzioni.

Il SO non è a conoscenza della natura di un programma, incluse le funzioni usate nel codice, ma è a conoscenza solo delle system call richiamate. Il resto dell'esecuzione è sotto il controllo del programma quindi le funzioni di un programma possono essere processi separati oppure possono costituire una parte di codice in un singolo processo.

5.1.3 Processi figli

Quando viene eseguito un programma, il kernel crea un processo, che possiamo chiamare **processo primario**. Questo può effettuare chiamate di sistema per creare altri processi, che diventano suoi **processi figli**. Questi a loro volta possono creare altri processi formando così un *albero di processi* la cui radice è il processo primario.

Solitamente, un processo crea uno o più figli in modo tale da delegare ad ognuno di essi una parte del suo lavoro; questa tecnica prende il nome di **multitasking** e presenta tre benefici:

1. *Speedup dell'elaborazione (maggiore velocità di esecuzione)*

Diminuzione del tempo di esecuzione dell'applicazione grazie alla creazione di processi figli. Se il processo primario non creasse processi figli, eseguirebbe le operazioni sequenzialmente; invece creando i processi figli, questi eseguono concorrentemente le operazioni.

2. *Priorità per le funzioni critiche*

Molti SO consentono a un processo genitore di assegnare priorità ai processi figli. Un'applicazione real-time può assegnare una priorità alta a un processo figlio che deve eseguire una funzione critica in modo tale da soddisfare i suoi requisiti di risposta.

3. *Proteggere un processo genitore dagli errori*

Il kernel può terminare un processo figlio in caso di errore, ma il padre resta protetto e può avviare un'azione di recupero.

Per facilitare l'uso dei processi figli, il kernel fornisce funzioni per creare un processo figlio e assegnargli una priorità, terminare un processo figlio e determinare lo stato di un processo figlio. Inoltre permette la condivisione, la comunicazione e la sincronizzazione dei processi figli.

5.1.4 Concorrenza e parallelismo

L'esecuzione di un programma può essere velocizzata utilizzando sia il *parallelismo* che la *concorrenza*.

Il **parallelismo** si riferisce alla caratteristica di verificarsi allo stesso tempo, dunque due processi sono eseguiti in parallelo se sono eseguiti nello stesso tempo. Questa tecnica di programmazione è realizzata usando più CPU.

La **concorrenza** è un'illusione di parallelismo, infatti c'è solo l'illusione che i processi sono eseguiti nello stesso tempo. In realtà, il singolo processore, dotato di un'alta velocità di elaborazione, dà all'utente solo la sensazione di eseguire più processi nello stesso tempo, ma, in effetti, esegue i processi uno per volta.

Quindi il SO può servire un processo e alcuni dei suoi processi figli in maniera concorrente eseguendo, sulla stessa CPU, le istruzioni di ognuno di essi per un certo periodo di tempo, o servirli in parallelo eseguendo le loro istruzioni su diverse CPU allo stesso tempo. Il risultato, cioè il raggiungimento dell'obiettivo comune, è raggiunto adoperando le tecniche di *sincronizzazione dei processi* messe a disposizione dal SO.

5.2 Implementazione dei processi

Uno dei compiti del kernel è quello di controllare l'esecuzione dei processi. In pratica deve allocare le risorse a un processo, proteggerlo da interferenze e assicurarsi che ottenga l'uso della CPU per completare la sua esecuzione.

Il kernel viene attivato in seguito a un *evento*, cioè il verificarsi di una situazione che porta alla generazione di un interrupt o ad una system call. A questo punto effettua quattro funzioni:

- **Salvataggio del contesto:** salva lo stato della CPU e le informazioni riguardanti il processo interrotto.
- **Gestione dell'evento:** analizza la condizione che ha generato l'interrupt o la system call ed effettua le azioni appropriate.
- **Scheduling:** seleziona il prossimo processo da eseguire sulla CPU.
- **Dispatching:** impostare il controllo della CPU (o l'accesso alle risorse) per il processo schedulato e caricare il suo stato salvato della CPU per iniziare o proseguire l'esecuzione e modifica lo stato del processo in *running*.

5.2.1 Attività di un processo: stati del processo e transizioni di stato

Un SO utilizza la nozione di *stato di un processo* per tener traccia dell'attività di un processo.

Lo **stato di un processo** è l'indicatore che descrive la natura dell'attività corrente di un processo.

Una **transizione di stato** per un processo consiste nel cambiamento di stato. Di solito è causata dal verificarsi di un evento. E' il kernel che, una volta verificatosi l'evento, modifica lo stato dei processi interessati.

Sono due i diagrammi usati per descrivere l'attività di un processo.

DIAGRAMMA A 4 STATI

Molti SO usano i quattro stati fondamentali descritti nella tabelle seguente:

Stato	Descrizione
<i>Running</i>	Una CPU sta eseguendo le istruzioni di un processo.
<i>Blocked</i>	Il processo deve aspettare finché non viene soddisfatta una sua richiesta per una risorsa o finché non si verifica uno specifico evento.
<i>Ready</i>	Il processo richiede l'uso della CPU per continuare la sua esecuzione; tuttavia, non è stato ancora eseguito il dispatch.
<i>Terminated</i>	L'esecuzione del processo, ovvero, l'istanza del programma che rappresenta, è stata completata correttamente o è stata terminata dal kernel.

Un processo è *bloccato* (*blocked*) quando è in attesa che una risorsa da lui richiesta gli venga allocata o quando è in attesa del verificarsi di un particolare evento.

Un processo è *pronto* (*ready*) quando può essere schedulato, cioè quando la richiesta è soddisfatta o quando si è verificato l'evento che stava attendendo.

Un processo è *in esecuzione* (*running*) quando la CPU sta eseguendo le istruzioni del processo stesso.

Un processo è *terminato* (*terminated*) quando viene portata a termine la sue esecuzione o se, per qualche motivo, viene terminato dal kernel.

Un computer che ha una sola CPU può avere un solo processo nello stato di *running*, ma più processi negli stati *blocked*, *ready* e *terminated*.



La figura mostra le fondamentali transizioni di stato per un processo.

Da notare il fatto che un processo può entrare negli stati *ready*, *running* e *blocked* anche più volte, mentre può entrare nello stato di *terminazione* una sola volta.

(NEW) → READY: un nuovo processo entra nello stato *ready* dopo che le risorse richieste sono state allocate. In questo stato ci sono tutti i processi che si trovano in memoria centrale.

N.B. per andare in esecuzione, il processo deve essere caricato in memoria centrale.

READY → RUNNING: un processo può andare nello stato *running* solo se in precedenza si trovava nello stato *ready* e ci va non appena viene completato il dispatching. In pratica un processo va in esecuzione non appena gli viene passato il controllo della CPU. Dallo stato *running* può raggiungere tutti gli altri stati.

RUNNING → TERMINAZIONE: un processo passa dallo stato *running* a quello di *terminazione* quando viene portata a termine l'esecuzione del programma. Le cause che portano alla terminazione di un programma sono molteplici (es: auto-terminazione, terminazione richiesta dal processo padre, eccesso di utilizzo di una risorsa, condizioni anomale durante l'esecuzione, interazione non corretta con altri processi). Quando un processo è terminato, vengono liberate tutte le risorse che gli erano state assegnate.

RUNNING → READY: un processo passa dallo stato *running* a quello *ready* quando viene prelazionato poiché il kernel decide di schedulare un altro processo (es: un processo a priorità più alta va nello stato *ready* oppure la time-slice del processo si esaurisce).

RUNNING → BLOCKED: un processo passa dallo stato *running* a quello *blocked* quando effettua una system call per richiedere l'uso di una risorsa o quando rimane in attesa di un evento. Un processo bloccato si trova in memoria, ma non necessariamente in memoria centrale (ad esempio in memoria swap).

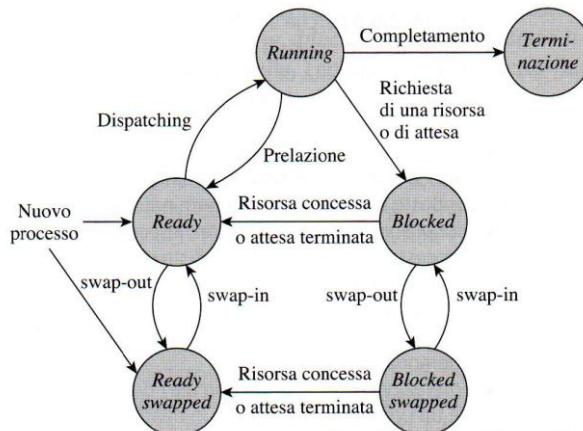
BLOCKED → READY: quando la richiesta del processo viene soddisfatta o quando si verifica l'evento, il processo passa dallo stato *blocked* a quello *ready*.

DIAGRAMMA A 6 STATI

Un kernel deve prevedere, però, altri due stati che non sono stati descritti precedentemente. Infatti un processo può anche essere *sospeso* (*swapped*). Il kernel può definire più stati *sospesi* e impostare il processo nello stato sospeso opportuno.

Un processo è *ready swapped* quando viene swappato sul disco e rimane in attesa. Prima che possa riprendere l'esecuzione deve essere riportato in memoria.

Un processo è *blocked swapped* se l'utente specifica che il processo non deve essere considerato schedulabile per un certo periodo di tempo (è comunque swappato sul disco).



La figura mostra gli stati del processo e le transizioni conformi all'organizzazione che prevede anche i due stati sopracitati.

(NEW) → READY: un nuovo processo entra nello stato *ready* quando è pronto per essere eseguito e c'è spazio in memoria centrale.

(NEW) → READY SWAPPED: un nuovo processo entra nello stato *ready swapped* quando è pronto per eseguito ma non c'è spazio in memoria centrale. Viene salvato sul disco (nell'area di swap), quindi non è ancora selezionabile per prendere il controllo della CPU.

READY SWAPPED → READY: un processo che si trova nello stato di *ready swapped*, prima di poter ottenere l'uso della CPU, deve prima essere "swappato" in memoria centrale mediante un'operazione di swap-in.

READY → READY SWAPPED: se il sistema ha l'esigenza di liberare la memoria per l'esecuzione di altri processi, può swappare alcuni processi sul disco mediante un'operazione di swap-out.

BLOCKED → BLOCKED SWAPPED: analoga situazione a quella in alto (*ready → ready swapped*).

BLOCKED SWAPPED → READY SWAPPED: questa transizione si verifica se viene soddisfatta la richiesta per cui il processo è in attesa o se si verifica l'evento. Ciò non toglie che il processo si trova sempre sul disco nell'area di swap e non in memoria centrale.

5.2.2 Salvataggio del contesto, scheduling e dispatching

Abbiamo detto in precedenza che un processo che si trova nello stato *running* può essere interrotto per varie ragioni (viene prelazionato, attende un evento, richiede delle risorse). Quando un processo viene interrotto, una funzione si occupa del **salvataggio del contesto**, cioè salva lo stato del processo e della CPU relativo al processo nel PCB. Questa funzione, inoltre, imposta lo stato del processo a *ready* oppure a *blocked*.

La **funzione di scheduling** utilizza le informazioni di stato salvate nel PCB per attribuire l'uso della CPU ad uno dei processi *ready*. Una volta selezionato, passa l'ID del processo selezionato alla **funzione di dispatching**. Questa funzione imposta il contesto del processo selezionato, modifica il suo stato in *running* e carica lo stato della CPU salvato dal PCB alla CPU.

COMMUTAZIONE DI CONTESTO (CONTEXT SWITCH)

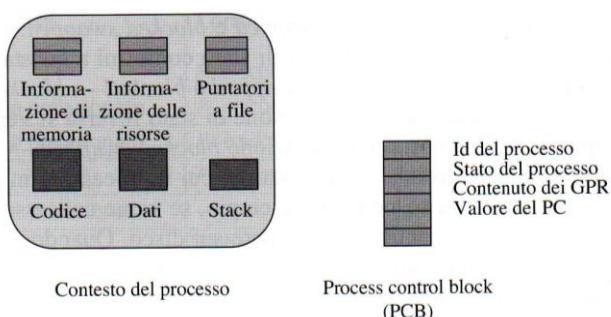
La **commutazione di contesto** o **context switch** è un particolare stato del SO durante il quale avviene il cambiamento del processo correntemente in esecuzione sulla CPU. Questo permette a più processi di condividere una stessa CPU, permettendo di eseguire più programmi contemporaneamente.

Prima di tutto è necessario salvare lo stato del processo correntemente in esecuzione, tra cui il PC (Program Counter) ed il contenuto dei registri generali, in modo che l'esecuzione potrà essere ripresa in seguito. Queste informazioni sullo stato del processo vengono salvate nel PCB. Successivamente lo scheduler sceglierà un processo tra quelli *ready*, in base alla propria politica di scheduling, e accederà al suo PCB per ripristinare il suo stato nel processore, in maniera inversa rispetto alla fase precedente.

Tutte le informazioni che è necessario salvare e ripristinare durante la commutazione prendono il nome di *informazione di stato di un processo*. L'overhead dovuto alla commutazione dipende dalla dimensione dell'informazione di stato del processo.

5.2.3 Il contesto del processo e il process control block

Abbiamo introdotto il concetto di PCB.



La gestione dei processi avviene nel kernel, quindi è il kernel che si occupa di allocare le risorse al processo e di schedularlo per l'uso della CPU. Di conseguenza, il kernel vede il processo composto di due parti:

- **contesto del processo**
- **descrittore (Process Control Block)**

CONTESTO DEL PROCESSO

Il contesto consiste in una serie di informazioni tra cui il codice, i dati, lo stack, le informazioni sulla memoria allocata al processo e le informazioni sulle risorse che utilizza.

In particolare, il contesto del processo si compone delle seguenti parti:

1. *Spazio di indirizzamento del processo*: codice, dati e stack del processo.
2. *Informazione di allocazione della memoria*: informazione relativa all'area di memoria allocata al processo. viene usata dal MMU per la gestione della memoria.
3. *Stato delle attività di elaborazione del file*: informazione relativa ai file usati.
4. *Informazione relativa all'interazione col processo*: informazione necessaria per controllare l'interazione del processo con gli altri processi.
5. *Informazione relativa alle risorse*: informazione riguardante le risorse allocate al processo.
6. *Informazioni varie*: varie informazioni utili per il corretto funzionamento del processo.

PROCESS CONTROL BLOCK

Per gestire più processi, ad ognuno di essi viene associato un **descrittore (Process Control Block)**.

Esso contiene lo stato di un processo e lo stato della CPU relativo al processo se la CPU non sta eseguendo le sue istruzioni.

In particolare, il PCB di un processo contiene tre tipi di informazione:

- *l'informazione relativa all'identificazione*, come per esempio l'ID del processo stesso, l'ID del genitore e l'ID dei figli.
- *l'informazione circa lo stato del processo*, ovvero il suo stato e il contenuto del PSW e dei registri GPR.
- *l'informazione necessaria per controllare la sua esecuzione*, cioè la priorità e l'interazione con gli altri processi.

Inoltre, visto che i descrittori sono mantenuti dal SO in una tabella dei processi, ognuno di essi contiene anche un puntatore usato dal kernel per creare tale lista, necessaria per la schedulazione.

Nella seguente tabella vengono descritti tutti i campi del PCB:

Campo del PCB	Contenuto
Id del processo	L'id univoco assegnato al processo al momento della creazione.
Id del genitore e dei figli	Questi id sono usati per la sincronizzazione dei processi, tipicamente per consentire a un processo di verificare se un processo figlio ha terminato la sua esecuzione.
Priorità	La priorità è generalmente un valore numerico. Al momento della creazione, al processo viene assegnata una priorità. Il kernel può cambiare la priorità dinamicamente in base alla natura del processo (CPU-bound o I/O-bound), al tempo di attività e alle risorse utilizzate (solitamente il tempo di CPU).
Stato del processo	Lo stato corrente del processo.
PSW	Questa è un'istantanea, ovvero un'immagine del PSW eseguita l'ultima volta che il processo è stato bloccato o prelazionato. Il caricamento di questa immagine nel PSW ripristina l'esecuzione del processo (vedi Figura 2.2 per i campi del PSW).
GPR	Il contenuto dei registri general purpose salvati l'ultima volta che il processo è stato bloccato o prelazionato.
Informazione sugli eventi	Per un processo nello stato <i>blocked</i> , questo campo contiene l'informazione relativa all'evento per il quale il processo è in attesa.
Informazioni sui segnali	Informazione relativa ai gestori dei segnali (vedi Paragrafo 5.2.6).
Puntatore al PCB	Questo campo è utilizzato per creare una lista di PCB, necessaria per la schedulazione.

5.2.4 Gestione degli eventi

Durante il funzionamento di un SO possono verificarsi vari eventi:

1. *Evento per la creazione di un processo*: viene creato un nuovo processo
2. *Evento per la terminazione di un processo*: un processo termina la sua esecuzione
3. *Evento timer*: si verifica un interrupt del timer
4. *Evento per la richiesta di una risorsa*: un processo effettua la richiesta per una risorsa
5. *Evento per il rilascio di una risorsa*: un processo rilascia una risorsa
6. *Evento per la richiesta di avvio di I/O*: un processo richiede l'avvio di un'operazione di I/O
7. *Evento per il completamento di I/O*: un'operazione di I/O viene completata
8. *Evento per l'invio di un messaggio*: un processo invia un messaggio a un altro processo
9. *Evento per la ricezione di un messaggio*: un processo riceve un messaggio
10. *Evento per l'invio di un segnale*: un processo invia un segnale a un altro processo
11. *Evento per la ricezione di un segnale*: un processo riceve un segnale
12. *Un interrupt da programma*: l'istruzione corrente nel processo *running* non è eseguita correttamente
13. *Evento per il malfunzionamento dell'hardware*: un'unità hardware del computer ha causato un malfunzionamento

Gli eventi dovuti al timer, al completamento di un'operazione di I/O e al malfunzionamento dell'hardware sono causati da situazioni esterne al processo *running*.

EVENT CONTROL BLOCK (ECB)

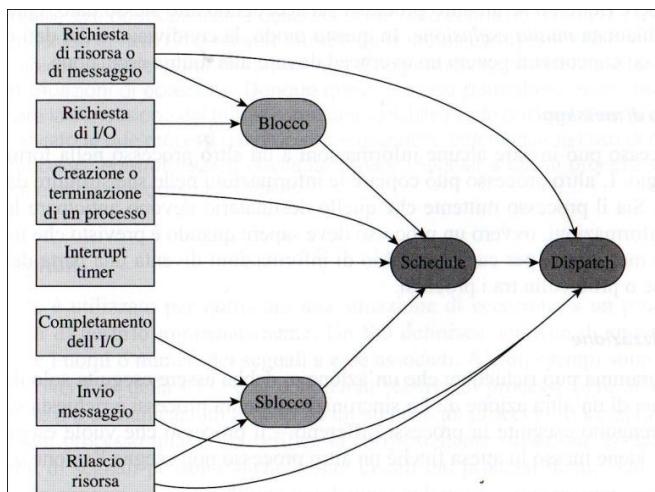
Quando si verifica un evento, il kernel deve trovare il processo il cui stato è influenzato dall'evento. L'operazione è effettuata mediante una ricerca nei campi delle *informazioni relative agli eventi* dei PCB dei processi. Questa ricerca è costosa in termini di tempo, per cui i SO usano vari schemi per velocizzarla. Uno di questi consiste nell'utilizzo dell'**Event Control Block (ECB)**.

Un ECB contiene tre campi:

- campo *descrizione dell'evento*, che descrive un evento
- campo *ID del processo*, che contiene l'ID del processo in attesa dell'evento
- campo *puntatore all'ECB*, che è necessario per inserire l'ECB di un processo nella lista appropriata, in quanto il kernel può mantenere più liste separate di ECB per ogni classe di evento

Quando si verifica un evento, il kernel esamina la lista degli ECB appropriata per trovare un ECB con una descrizione dell'evento corrispondente. Il campo ID indica quale processo è in attesa dell'evento.

GESTIONE DELL'EVENTO



La figura illustra le azioni del kernel per la gestione dell'evento descritte in precedenza.

L'azione **block** modifica sempre lo stato del processo chiamante da *ready* a *blocked*.

L'azione **unblock** cerca un processo la cui richiesta può essere soddisfatta e cambia lo stato da *blocked* a *ready*.

Una system call per la richiesta di una risorsa implica un'azione di *block* (seguita dallo scheduling e dal dispatching) se la risorsa non può essere allocata direttamente al processo che la richiede. Invece, se la risorsa può essere allocata direttamente, allora l'azione di blocco non viene effettuata e il processo viene semplicemente sottoposto a dispatching.

Quando un processo rilascia una risorsa, nel caso in cui un altro processo sia in attesa della risorsa rilasciata, viene eseguita un'azione di *unlock* (seguita da scheduling e dispatching), poiché il processo sbloccato può avere una priorità maggiore del processo che ha rilasciato la risorsa. Anche in questo caso, lo scheduling non viene effettuato se in conseguenza dell'evento nessun processo è stato sbloccato.

5.2.5 Condivisione, comunicazione e sincronizzazione tra processi

I processi di un'applicazione hanno bisogno di interagire l'uno con l'altro per un obiettivo comune.

Vi sono quattro tipi di interazione tra processi:

Tipo di interazione	Descrizione
Condivisione dei dati	I dati condivisi possono diventare inconsistenti se diversi processi li modificano allo stesso tempo. Per questo motivo i processi devono interagire per decidere quando è possibile per un processo modificare o usare i dati condivisi in modo sicuro.
Scambio di messaggi	I processi scambiano informazioni inviando messaggi l'uno all'altro.
Sincronizzazione	Per ottenere un obiettivo comune, i processi devono coordinare le proprie attività.
Segnali	Un segnale è usato per comunicare a un processo l'occorrenza di una situazione di eccezione.

5.2.6 Segnali

Un segnale è utilizzato per notificare una situazione di eccezione a un processo e consentirgli di gestirlo immediatamente. Un SO definisce una lista di situazioni di eccezione e i nomi o numeri dei segnali a esse associate. Il kernel invia un segnale a un processo quando si verifica la corrispondente situazione di eccezione. Alcuni tipi di segnali possono anche essere inviati dai processi utente.

Un segnale inviato a un processo a causa di una condizione nel suo funzionamento è chiamato *segnale sincrono*, mentre quello inviato a causa di qualche altra condizione è chiamato *segnale asincrono*.

Per utilizzare i segnali, un processo effettua una chiamata di sistema con la quale specifica una routine da eseguire alla ricezione di uno specifico segnale; questa routine è chiamata *signal handler*. Se un processo non specifica un signal handler per un segnale, il kernel esegue un *default handler* che effettua alcune azioni standard (come la terminazione del processo).

In un SO la gestione del segnale nel processo è implementata con le stesse modalità della gestione degli interrupt. In ogni processo, l'area di memorizzazione dei vettori dei segnali contiene un vettore dei segnali per ogni tipo di segnale, che a sua volta contiene l'indirizzo di un signal handler. Quando un segnale è inviato a un processo, il kernel accede all'area di memorizzazione dei vettori dei segnali per verificare se è stato specificato un signal handler per quel segnale. In caso affermativo, passa il controllo al signal handler; altrimenti, esegue il default handler per quel segnale.

5.3 Thread

Le applicazioni usano processi concorrenti per velocizzare la loro esecuzione. Tuttavia, la commutazione tra i processi all'interno di un'applicazione genera un elevato overhead dovuto alla quantità di informazione da salvare e ripristinare ad ogni commutazione. Per questo motivo i progettisti dei sistemi operativi hanno sviluppato un modello alternativo di esecuzione, chiamato *thread*, che favorisce la concorrenza all'interno di un'applicazione con un ridotto overhead.

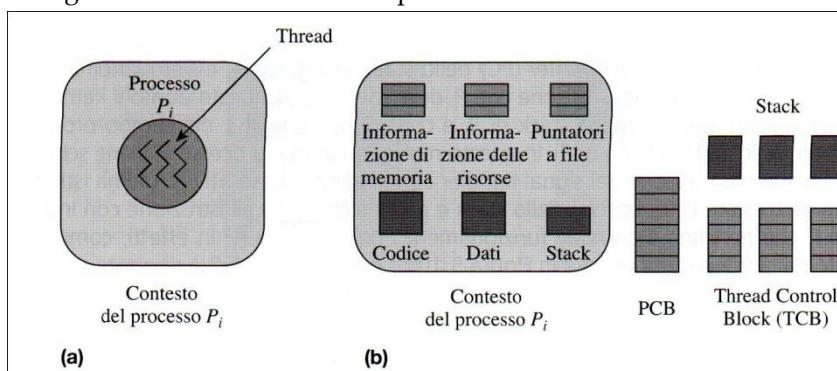
Un **thread** rappresenta un modello alternativo di esecuzione di un programma che usa le risorse di un processo. Un processo può contenere più thread, ciascuno dei quali evolve in modo logicamente separato dagli altri thread.

Per comprendere meglio la nozione di thread analizziamo l'overhead dovuto alla commutazione tra processi per determinare un possibile risparmio di tempo. L'overhead dovuto alla commutazione tra processi ha due componenti:

1. *Overhead relativo all'esecuzione*: lo stato della CPU del processo in esecuzione deve essere salvato e lo stato della CPU del nuovo processo deve essere caricato nei registri della CPU. Questo overhead è inevitabile ed è generato anche se si utilizzano i thread.
2. *Overhead dovuto all'uso delle risorse*: anche il contesto del processo deve essere commutato. La maggior parte di queste informazioni aggiunge overhead alla commutazione di processo.

Un thread differisce da un processo nel fatto che a esso non sono allocate risorse. Questa differenza riduce l'overhead della commutazione tra thread rispetto all'overhead della commutazione tra processi.

Un processo crea un thread mediante una system call. Il thread non possiede risorse proprie, quindi non ha un contesto; viene eseguito usando il contesto del processo e in tal modo accede alle risorse del processo.



La figura illustra la relazione tra thread e processi.

Nella figura a sinistra, il processo (indicato dal cerchio) ha tre thread (indicati da linee ondulate).

Nella figura a destra, il processo ha un contesto e un PCB; ogni thread è un'istanza del programma, per cui ha un proprio stack e un proprio *Thread Control Block* (TCB), analogo al PCB e contenente le seguenti informazioni:

- informazioni per la schedulazione dei thread: thread ID, priorità, stato;
- stato della CPU: contenuto della PSW e dei registri GPR;
- puntatore al PCB del processo genitore;
- puntatore al TCB, usato per creare le liste di TCB per la schedulazione.

ATTIVITA' DI UN THREAD: STATI DEI THREAD E TRANSIZIONI DI STATO

Fatta eccezione per il fatto che i thread non hanno allocate risorse proprie, i thread e i processi sono analoghi, dunque gli stati dei thread e le transizioni di stato sono analoghi agli stati e alle transizioni di stato dei processi.

Le uniche differenze sono relative allo stato *blocked* e *swapped*.

Si è detto in precedenza che un processo passa dallo stato *running* a quello *blocked* quando effettua una system call per richiedere l'uso di una risorsa o quando rimane in attesa di un evento. Nel caso dei thread la richiesta per l'uso di una risorsa non può verificarsi poiché non effettua nessuna richiesta di risorsa; tuttavia, il processo può entrare nello stato *blocked* a causa della seconda condizione.

Inoltre non esiste lo stato *swapped* (sia *ready* che *blocked*) perché la memoria è una risorsa del processo.

VANTAGGI DEI THREAD RISPETTO AI PROCESSI

Abbiamo già visto che il primo vantaggio dei thread rispetto ai processi consiste nel *minor overhead relativamente alla creazione e alla commutazione*. Ci sono però anche altri vantaggi che, in varie situazioni, fanno preferire l'uso dei thread all'uso dei processi.

Uno di questi è quello di avere una *comunicazione più efficiente*. In pratica, visto che i thread (diversamente dai processi) condividono lo spazio di indirizzamento del processo genitore, possono comunicare tra loro attraverso dati condivisi anziché mediante messaggi, evitando in questo modo l'overhead di comunicazione dovuto alle system call.

Un altro vantaggio rispetto ai processi è la *progettazione semplificata*. Infatti l'uso dei thread può semplificare la progettazione e la codifica delle applicazioni che servono le richieste concorrentemente (es: prenotazioni dei voli online). La creazione e la terminazione dei thread è più efficiente rispetto alle medesime operazioni sui processi; tuttavia, il suo overhead può causare un decadimento delle prestazioni del server nel caso in cui i client effettuassero un numero elevato di richieste. Per questo motivo si ricorre ad un'organizzazione chiamata *thread pool* che evita questo overhead: consiste nel riutilizzare i thread invece di distruggerli dopo aver soddisfatto le richieste evitando così l'overhead dovuto alla creazione e alla terminazione dei thread.

CODIFICA PER L'UTILIZZO DEI THREAD

I thread assicurano la correttezza dei dati condivisi e la sincronizzazione. Però ciò avviene se l'applicazione che utilizza i thread è codificata in *thread safe*; ciò non accade se è codificata in *thread unsafe*.

Nelle applicazioni *thread safe* i dati globali sono "protetti" dall'utilizzo della mutua esclusione, dunque non è possibile produrre risultati inconsistenti.

5.3.1 Thread di livello kernel, di livello utente e ibridi

Tre sono i modelli di thread utilizzati: *thread di livello kernel*, *thread di livello utente* e *thread ibridi*. Ognuno di essi ha differenti implicazioni sull'overhead della commutazione, sulla concorrenza e sul parallelismo.

THREAD DI LIVELLO KERNEL

Un thread di livello kernel è implementato dal kernel, dunque la creazione, la terminazione e il recupero dello stato di un thread kernel sono effettuati mediante system call.

L'overhead dovuto alla commutazione, nel caso di thread di livello kernel, risulta 10 volte più veloce rispetto alla commutazione tra processi.

VANTAGGI: un thread di livello kernel è come un processo eccetto che ha una quantità inferiore di informazioni di stato. Questa similarità conviene ai programmatore visto che la programmazione dei thread non differisce molto dalla programmazione dei processi.

SVANTAGGI: la commutazione dei thread è effettuata dal kernel dunque viene generato overhead anche se il thread interrotto e il thread selezionato appartengono allo stesso processo. C'è da dire che l'overhead generato è comunque inferiore rispetto a quello generato dai processi.

THREAD DI LIVELLO UTENTE

I thread di livello utente sono implementati da una *libreria di thread*, che viene linkata al codice del processo. In questo tipo di thread il kernel non viene coinvolto ed è la libreria stessa che gestisce l'alternanza dell'esecuzione dei thread nel processo. In questo modo, il kernel non è a conoscenza della presenza dei thread di livello utente in un processo; il kernel vede solo il processo.

VANTAGGI: la sincronizzazione e la schedulazione dei thread sono implementate dalla libreria. Questa organizzazione evita l'overhead della system call per la sincronizzazione dei thread, per cui l'overhead dovuto alla commutazione dei thread potrebbe essere 10 volte più veloce rispetto ai thread di livello kernel. Inoltre questa organizzazione consente a ogni processo di usare una politica di scheduling che meglio si adatta alla sua natura.

SVANTAGGI: usando thread di questo tipo, il kernel non conosce la differenza tra thread e processo, per cui se un thread si bloccasse su una system call, il kernel bloccherebbe il processo genitore. In effetti, tutti i thread del processo sarebbero bloccati finché non sarebbe rimossa la causa del blocco. Per facilitare questa situazione, un SO dovrebbe rendere disponibile una versione non bloccante di ogni system call.

MODELLO DEI THREAD IBRIDO

Un modello del genere implementa sia i thread di livello utente che i thread di livello kernel e anche un metodo per associare i thread di livello utente ai thread di livello kernel. Come risultato si possono ottenere differenti combinazioni caratterizzate da ridotto overhead di commutazione dei thread di livello utente ed elevata concorrenza e parallelismo dei thread di livello kernel.

La libreria di thread crea thread utente in un processo e, a ogni thread utente, associa un *thread control block* (TCB). Il kernel crea i thread kernel in un processo e, a ogni thread kernel, associa un *kernel thread control block* (KTCB).

Ci sono tre metodi per associare i thread di livello utente ai thread di livello kernel.

1. *molti a uno* – il kernel crea un singolo thread kernel nel processo e tutti i thread utente creati nel processo sono associati con l'unico thread kernel. L'effetto creato è simile ai semplici thread utente. I thread utente non possono essere paralleli, l'overhead è basso e il blocco di un thread porta al blocco di tutti i thread nel processo.
2. *uno a uno* – ogni thread utente è mappato permanentemente su un thread kernel. L'effetto è simile ai semplici thread kernel. I thread possono essere eseguiti in parallelo su diverse CPU, la commutazione produce un overhead più elevato perché è eseguita dal kernel, il blocco di un thread non blocca tutti gli altri thread perché i thread sono mappati su diversi thread kernel.
3. *uno a molti* – ad ogni thread utente è permesso di essere mappato su differenti thread kernel in momenti diversi. Questo modello consente di sfruttare il parallelismo e di generare poco overhead. Tuttavia, l'implementazione risulta più complessa.

5.4 Casi di studio

5.4.1 Processi in Unix

Unix utilizza due strutture dati per memorizzare i dati di controllo relativi ai processi:

- **Struttura proc:** memorizza i dati relativi alla schedulazione.
In particolare contiene ID processo, stato del processo, priorità, relazioni con altri processi, descrittore dell'evento per il quale un processo bloccato è in attesa, maschera per la gestione dei segnali, informazioni relative alla gestione della memoria
- **U-area:** contiene i dati relativi all'allocazione delle risorse e alla gestione dei segnali.
La user-area contiene un PCB per conservare lo stato di un processo bloccato, il puntatore alla struttura *proc*, gestore dei segnali, file aperti, directory corrente

Queste due strutture mantengono l'informazione analoga alla struttura PCB vista nei paragrafi precedenti.

TIPI DI PROCESSI

In Unix esistono due tipi di processi: *processi utente* e *processi kernel*.

Un *processo utente* esegue operazioni per l'utente e dipende dal terminale di controllo. Quando un utente avvia un programma, il kernel crea il processo primario, che a sua volta può creare processi figli.

Un *processo daemon* esegue operazioni di sistema e non dipende dal terminale di controllo. Viene eseguito in background, talvolta rimanendo in esecuzione durante tutto il funzionamento del sistema.

Un *processo kernel* esegue il codice del kernel e vengono eseguiti anch'essi in background. Sono creati all'avvio del sistema.

CREAZIONE E TERMINAZIONE DI UN PROCESSO

La system call *fork* crea un processo figlio, imposta il suo contesto e restituisce l'ID del figlio.

Quando crea un processo alloca una struttura *proc* per il processo appena creato, imposta il suo stato a *ready* e infine alloca la *u-area* per il processo. La relazione genitore-figlio viene tracciata mediante la struttura *proc*.

Il contesto del figlio è una copia del contesto del padre, quindi il figlio esegue lo stesso codice del genitore. La chiamata a *fork* restituisce uno 0 nel processo figlio, e questa differenza col padre, può far eseguire al figlio e al padre due parti del codice differenti. Inoltre un processo figlio può usare la system call *exec* per caricare un altro programma da eseguire.

In definitiva, un figlio può scegliere se eseguire il codice contenuto nel contesto del padre (o solo una parte di esso) oppure un altro programma.

Un processo può *terminare* se stesso richiamando la system call *exit*. L'argomento di questa system call è il codice che indica lo stato di terminazione, detto *status code*. La *exit* invia anche un segnale al processo padre. Se il processo che richiama la *exit* aveva dei processi figli, questi diventano figli di *init*.

ATTESA DELLA TERMINAZIONE DI UN PROCESSO

Un processo può anche attendere la terminazione di un altro processo mediante la system call *wait*. Se uno dei processi figli del processo che ha invocato la *wait* termina, al processo padre ritorna lo stato di terminazione del processo figlio terminato.

Se più di un processo figlio termina, il padre deve usare necessariamente più *wait* per ricevere lo stato di terminazione dei processi figli terminati.

CAPITOLO 6. La gestione della CPU – Scheduling

Una politica di scheduling decide a quale processo debba essere assegnata la CPU in un certo momento. Per questo motivo, le decisioni dello scheduler influenzano sia il servizio utente che le prestazioni del sistema.

In questo capitolo saranno discusse le tre tecniche che usa lo scheduler per ottenere la combinazione ottimale tra servizio utente e prestazioni del sistema:

- 1) Assegnazione delle priorità
- 2) Riordino delle richieste
- 3) Variazione della time-slice

6.1 Scheduling: terminologia e concetti

Con la multiprogrammazione, diversi processi competono per l'uso della CPU: cioè quando in memoria ci sono più processi nello stato di *ready*. La parte del sistema operativo che decide quale processo eseguire viene chiamata *scheduler* e l'algoritmo che usa è chiamato *algoritmo di schedulazione*.

Lo **scheduling** è quella parte di codice che si occupa di selezionare il processo che passerà dallo stato *ready* allo stato *running*. Per effettuare questa scelta, lo scheduler considera una lista di richieste in attesa di essere elaborate e ne seleziona una per l'elaborazione.

Una volta che una richiesta viene elaborata dalla CPU, essa può essere completata oppure può essere prelazionata (interrotta e riportata nella lista delle richieste in attesa). In entrambi i casi, lo scheduler seleziona la prossima richiesta da elaborare.

Con *richiesta* si intende solitamente l'esecuzione di un job o di un processo. Gli eventi relativi ad una richiesta sono il suo *arrivo*, *ammissione*, *scheduling*, *prelazione* e *completamento*.

Un processo o un job è detto in *arrivo* quando viene sottomesso (inviato, avviato) dall'utente, ed è *ammesso* quando lo scheduler comincia a considerarlo nello scheduling.

Gli attuali SO usano politiche di scheduling molto più complesse rispetto a quelle viste nei capitoli precedenti, cioè lo scheduling Round-Robin e lo scheduling basato su priorità.

6.1.1 Indicatori di prestazioni

Descriviamo ora i concetti legati allo scheduling.

CONCETTI DI SCHEDULING RELATIVI ALLA RICHIESTA

Termine o concetto	Definizione o descrizione
Relativamente alla richiesta	
Tempo di arrivo	Istante in cui un utente invia un job o un processo.
Tempo di ammissione	Istante in cui il sistema comincia a considerare un job o un processo per lo scheduling.
Tempo di completamento	Istante in cui un job o un processo è terminato.
Deadline	Istante entro il quale un job o un processo deve essere terminato per rispettare il requisito di risposta di un'applicazione real-time.
Tempo di servizio	Il totale del tempo di CPU e quello di I/O richiesto da un job, processo o sottorichiesta per completare la sua operazione.
Prelazione	Deallocazione forzata della CPU da un job o da un processo.
Priorità	Una regola discriminante usata per selezionare un job o un processo quando molti job o processi attendono l'elaborazione.

La tabella elenca i termini e i concetti chiave relativi allo scheduling e alle richieste.

Il *tempo di servizio* di un processo è la somma del tempo di CPU e di quello di I/O da esso richiesto per completare la sua esecuzione.

La *deadline*, specificata solo nei SO real-time, è il tempo entro il quale l'esecuzione del processo dovrebbe essere completata.

Il *tempo di completamento* di un processo dipende dai suoi tempi di arrivo e di servizio e dal tipo di servizio che riceve dalla CPU.

CONCETTI DI SCHEDULING RELATIVI AL SERVIZIO PER L'UTENTE

Relativamente al servizio per l'utente: richieste individuali	
Superamento della deadline (deadline overrun)	La quantità di tempo per la quale il tempo di completamento di un job o un processo supera la sua deadline. Il deadline overrun può essere sia positivo sia negativo.
Condivisione equa	Una condivisione specifica del tempo di CPU che dovrebbe essere dedicato all'esecuzione di un processo o di un gruppo di processi.
Rapporto di risposta	Il rapporto $\frac{\text{tempo di attesa} + \text{tempo di servizio di un job o processo}}{\text{tempo di servizio del job o processo}}$
Tempo di risposta (r_t)	Il tempo tra la sottomissione di una sottorichiesta per l'elaborazione e il tempo in cui il suo risultato diventa disponibile. Questo concetto è applicabile ai processi interattivi.
Tempo impiegato per il completamento o tempo di turnaround (ta)	Il tempo che intercorre tra la sottomissione di un job o processo e il suo completamento da parte del sistema. Questo concetto è significativo solo per job non interattivi o processi.
Turnaround pesato (w)	Rapporto del tempo di turnaround di un job o processo e il suo tempo di servizio.
Relativamente al servizio per l'utente: servizio medio	
Tempo di risposta medio (\bar{r}_t)	La media dei tempi di risposta di tutte le sottorichieste elaborate dal sistema.
Tempo medio di turnaround (\bar{ta})	La media dei tempi di turnaround di tutti i job o processi elaborati dal sistema.

In un ambiente interattivo, un utente interagisce con un processo facendo una *sottorichiesta* al processo, e quest'ultimo risponde eseguendo delle azioni. Il *tempo di risposta* è il tempo che intercorre tra la sottomissione di una sottorichiesta e il momento in cui la sua elaborazione viene completata.

Il *tempo di completamento* (*tempo di turnaround*) differisce dal tempo di servizio in quanto include anche il tempo in cui il processo non è in esecuzione nella CPU e non sta eseguendo operazioni di I/O.

Il *completamento pesato* (*turnaround pesato*) mette in relazione il tempo di completamento di un processo con il suo tempo di servizio.

La *condivisione equa* è la condivisione del tempo di CPU che dovrebbe essere assegnata a un processo o a un gruppo di processi.

Il *rapporto di risposta* è un rapporto che descrive il ritardo nell'espletamento di un processo rispetto al suo tempo di servizio. Questo concetto è utile per evitare la starvation dei processi.

Il *deadline overrun* è la differenza tra il tempo di completamento e la deadline di un processo in un'applicazione real-time. Un valore negativo indica che il processo è stato completato prima della sua scadenza (deadline), mentre un valore positivo indica che la deadline non è stata rispettata.

CONCETTI DI SCHEDULING RELATIVI AL SISTEMA

Relativamente alle prestazioni	
Durata della schedulazione	Il tempo necessario per completare un insieme specifico di job o processi.
Throughput	Il numero medio di job, processi o sottorichieste completate da un sistema nell'unità di tempo.

Throughput e durata della schedulazione sono misure delle prestazioni del sistema.

Il *throughput* indica il numero medio di richieste completata nell'unità di tempo. Fornisce una base per confrontare due o più politiche di scheduling o per confrontare la stessa politica di scheduling in periodi diversi di tempo.

La *durata della schedulazione* indica la quantità di tempo totale utilizzata dalla CPU per completare un insieme di richieste. Serve anch'essa per valutare le prestazioni di una politica di scheduling, in particolare quando l'overhead non è trascurabile.

Un altro concetto non presente nella tabella è il *grado di multiprogrammazione*, ovvero il numero di processi presenti in memoria.

6.1.2 Tecniche fondamentali di scheduling

Gli scheduler usano tre tecniche fondamentali per fornire un buon compromesso tra prestazioni del sistema e servizio utente:

1. *Scheduling basato su priorità*: il processo in esecuzione è quello con la priorità più alta. Questo è assicurato selezionando il processo in stato *ready* a più alta priorità e prelazionando il processo in esecuzione quando un processo con priorità più alta diventa *ready*. Solitamente si assegna una priorità alta ai processi I/O bound in modo da avere un elevato throughput di sistema.
2. *Riordino delle richieste*: il riordino delle richieste implica il fatto che le richieste verranno espetate in un ordine diverso rispetto a quello di arrivo. Il riordino può migliorare sia il servizio utente che il throughput di sistema.
3. *Variazione della time-slice*: quando si fa uso del time-slicing, si ottengono tempi di risposta migliori quando vengono utilizzati valori inferiori per la time-slice; in questo modo viene ridotta anche l'efficienza della CPU a causa dei molteplici context switch.

6.1.3 Il ruolo della priorità

La priorità viene usata dallo scheduler quando più processi sono in attesa dell'uso della CPU. La priorità di una richiesta può essere determinata in funzione di vari parametri. E' detta *priorità dinamica* se alcuni dei suoi parametri cambiano durante l'elaborazione; altrimenti è detta *priorità statica*.

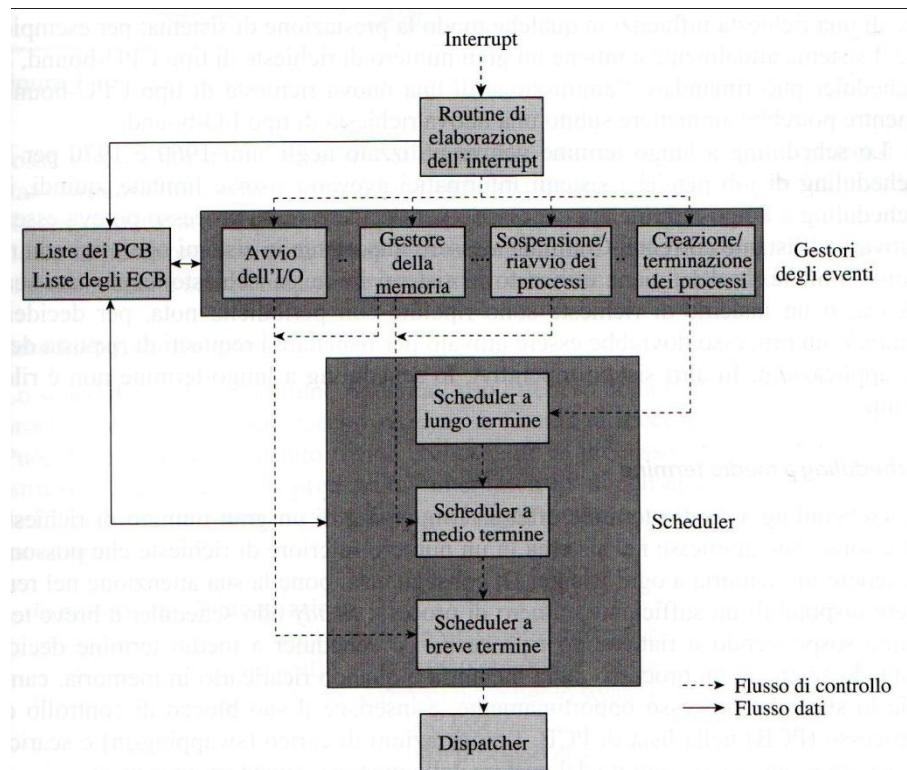
Lo scheduling basato su priorità ha lo svantaggio che una richiesta a bassa priorità potrebbe non essere mai espletata fintanto che arrivano richieste con priorità più alta. Questa situazione è detta *starvation*. Questa situazione potrebbe essere evitata incrementando la priorità di una richiesta che non è stata schedulata per un certo lasso di tempo. Questa tecnica prende il nome di *aging* delle richieste.

6.2 Classificazione delle attività di scheduling

Visto che un SO deve fornire una giusta combinazione tra prestazioni del sistema e servizio utente, occorrono *diversi* scheduler (cioè diversi algoritmi) ognuno dei quali può utilizzare una combinazione di diverse politiche di scheduling. Infatti, in un moderno SO, possono essere impiegati fino a tre scheduler:

1. *Scheduler a lungo termine*
2. *Scheduler a medio termine*
3. *Scheduler a breve termine*

6.2.1 Gestione degli eventi e schedulazione



La figura mostra una panoramica dello scheduling e delle relative azioni.

Ogni evento che richiede l'attenzione del kernel genera un interrupt. La routine di elaborazione dell'interrupt esegue una funzione di salvataggio del contesto e invoca un gestore di eventi. Il gestore di eventi analizza l'evento e cambia lo stato del processo per poi invocare, a seconda della necessità, lo scheduler a lungo, medio o breve termine.

Ad esempio, il gestore di eventi che crea un nuovo processo invoca lo scheduler a lungo termine, i gestori di eventi per la sospensione e il ripristino di processi invocano lo scheduler a medio termine se è esaurita la memoria. Comunque, la maggior parte dei gestori di eventi invocano lo scheduler a breve termine.

6.2.2 Scheduler a lungo, medio e breve termine

SCHEDULER A LUNGO TERMINE

Lo scheduler a lungo termine decide quale processo entra nella coda dei processi *ready* tra quelli che la richiedono, perciò controlla il grado di multiprogrammazione (il numero di processi presenti in memoria).

Lo scheduler a lungo termine può ritardare l'ammissione di una richiesta per due motivi:

- 1) non riesce ad allocare risorse sufficienti
- 2) l'ammissione della richiesta porterebbe ad un calo delle prestazioni del sistema

SCHEDULER A MEDIO TERMINE

Lo scheduler a medio termine gestisce la permanenza in memoria dei processi non in esecuzione. In pratica decide quali processi possono risiedere in memoria e quali sul disco.

Le operazioni di carico (swap in) e scarico (swap out) sono comunque effettuate dal gestore della memoria.

SCHEDULER A BREVE TERMINE

E' detto anche *dispatcher* o *scheduler*. Decide quale processo nello stato *ready* può andare in esecuzione. Inoltre, può anche decidere il tempo di utilizzo della CPU per il processo selezionato.

I momenti in cui può intervenire lo scheduler a breve termine sono quattro:

1. quando un processo passa dallo stato *running* allo stato *blocked* per selezionare quale altro processo *ready* andrà in esecuzione
2. quando un processo passa dallo stato *blocker* allo stato *ready* per determinare con quali altri processi *ready* compete
3. quando un processo passa dallo stato *running* allo stato *ready* (es. per una interruzione) per determinare se il processo tornerà in esecuzione subito oppure no
4. quando un processo termina per selezionare quale processo verrà eseguito

Nei casi 1 e 4 (*preemptive*) si deve scegliere un altro processo *ready*; nei casi 2 e 3 (*non preemptive*) si può scegliere un altro processo *ready*, oppure si può lasciare o mandare in esecuzione il processo che ha cambiato stato.

Quindi può essere invocato su eventi che possono causare il passaggio ad un altro processo (interrupt di clock, interrupt I/O, system call, segnalazioni tra processi).

6.2.3 Comportamento dei processi (cicli di CPU, cicli di I/O)

Ogni processo può essere visto come alternanza di cicli da parte della CPU e cicli da parte dell'I/O.

Quindi l'esecuzione di un processo è costituita da due fasi:

- esecuzione di istruzioni (*CPU burst*)
- attesa di eventi o operazioni di I/O (*I/O burst*)

Con *CPU burst* si indica una sequenza di operazioni svolte dalla CPU.

Con *I/O burst* si indica una sequenza di operazioni di I/O.

I processi si alternano tra questi due stati.

E' possibile operare una distinzione:

- i *processi CPU bound* hanno *CPU burst lunghi* (sono orientati ai calcoli), cioè richiedono molto tempo di CPU e poche operazioni di I/O
- i *processi I/O bound* hanno *CPU burst molto brevi* (sono orientati alle operazioni di I/O), cioè richiedono poco tempo di CPU ma molte operazioni di I/O

6.2.4 Politiche di scheduling della CPU

NON PREEMPTIVE vs PREEMPTIVE

Una prima classificazione è quella relativa alla presenza o meno della **prelazione**, che è la sospensione forzata dell'esecuzione di un processo per eseguirne un altro.

La *politica di scheduling preemptive* (con prelazione) dà la possibilità di interrompere il processo correntemente in esecuzione a favore di un altro processo. Essa seleziona un processo e lo lascia in esecuzione per una certa quantità di tempo massima. Se il processo, alla fine del tempo concessogli, è ancora in esecuzione, allora viene sospeso e lo scheduler seleziona un altro processo a cui attribuire l'uso della CPU.

Una politica del genere richiede un interrupt timer alla fine dell'intervallo di tempo per restituire il controllo della CPU allo scheduler.

La *politica di scheduling non preemptive* (senza prelazione) non dà la possibilità di interrompere un processo in esecuzione. Essa seleziona un processo e lo lascia in esecuzione finché non si *blocchia* (per attendere un evento o richiedere una risorsa) oppure finché non completa le sue operazioni rilasciando volontariamente la CPU.

SENZA PRIORITÀ vs CON PRIORITÀ

Un'altra classificazione è quella relativa alle politiche che si basano o meno sulla **priorità**.

Le politiche che usano la priorità selezionano i processi a cui attribuire l'uso della CPU proprio in base alla priorità acquisita da ciascuno di essi. Queste politiche sono necessarie nei sistemi real-time e nei sistemi interattivi.

Le politiche che non usano la priorità considerano i processi "equivalenti", cioè tutto sullo stesso piano senza privilegiare l'uno rispetto ad un altro. Queste politiche sono basate su strategie di ordinamento *First Come First Served*.

STATICHE vs DINAMICHE

Un'ultima classificazione è quella relativa alle politiche **statiche** e **dinamiche**.

Nelle *politiche statiche* ogni processo conserva nel tempo i suoi diritti di accesso alla CPU. In pratica le decisioni di schedulazione sono prese prima che il sistema inizi l'esecuzione.

Nelle *politiche dinamiche* ogni processo modifica i propri diritti di accesso alla CPU in base al comportamento passato o estrapolando quello futuro. In pratica le decisioni di schedulazione sono prese durante l'esecuzione dei processi.

6.2.5 Algoritmi di schedulazione

Esistono tre ambienti differenti, per i quali esistono determinati algoritmi di schedulazione:

- *batch*
- *interattivo*
- *real-time*

Nei *sistemi batch* non ci sono processi impazienti in attesa di una risposta veloce. Quindi sono accettabili sia algoritmi non preemptive, sia quelli preemptive con un periodo lungo per ogni processo. Questo approccio riduce gli scambi tra processi, migliorando le prestazioni.

Per i sistemi batch sono importanti il throughput, il tempo di turnaround e l'uso della CPU.

Politiche usate: FCFS, SJF (SNPF e SRTF).

Nei *sistemi interattivi* il prerilascio è essenziale per evitare che un processo si impossessi della CPU. La differenza con i sistemi real-time è che quelli interattivi sono d'uso generale e possono eseguire programmi arbitrari che non collaborano o sono persino dolosi.

Per i sistemi interattivi è importante il tempo di risposta.

Politiche usate: Round-Robin, Priorità, Code Multiple, HRRN

Nei *sistemi real-time* il prerilascio non è sempre necessario perché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e normalmente fanno il loro lavoro e si bloccano in fretta. Qui la differenza con i sistemi interattivi è che i sistemi real-time eseguono solo programmi che agevolano le altre applicazioni.

6.3 Politiche di scheduling nonpreemptive

Nello *scheduling non preemptive* un processo viene elaborato fino al completamento o fino a quando non si blocca. Per queste ragioni, lo scheduling non preemptive risulta molto semplice. Poiché una richiesta non è mai prelazionata, lo scheduler ha soltanto la funzione di riordinare le richieste per migliorare il servizio utente o le prestazioni del sistema.

Le tre politiche di scheduling non preemptive sono:

- Scheduling *First Come, First Served* (FCFS)
- Scheduling *Shortest Job First* (SJF) nella versione non preemptive (SNPF)
- Scheduling *Highest Response Ratio Next* (HRRN)

6.3.1 Scheduling First Come First Served (FCFS)

TIPO: lo scheduling FCFS è un algoritmo *senza prelazione*, senza priorità e statico.

FUNZIONAMENTO: i processi sono schedulati nell'ordine in cui giungono al sistema, cioè il primo processo ad essere eseguito è quello che per primo ha richiesto la CPU. I processi successivi vengono schedulati con lo stesso criterio non appena il processo in esecuzione completa le sue operazioni.

In pratica, i processi *ready* sono organizzati come una coda FIFO e i processi che richiedono la CPU vengono inseriti alla fine di questa coda.

CONCLUSIONI: questo tipo di algoritmo è semplice da implementare ma solitamente è poco efficiente, almeno considerando il tempo medio d'attesa. E' un algoritmo che privilegia i processi CPU bound, infatti questo algoritmo presenta il problema *effetto convoglio*: i processi che richiedono meno tempo di CPU (I/O bound) devono attendere che i processi che richiedono molto tempo di CPU (CPU bound) liberino la CPU.

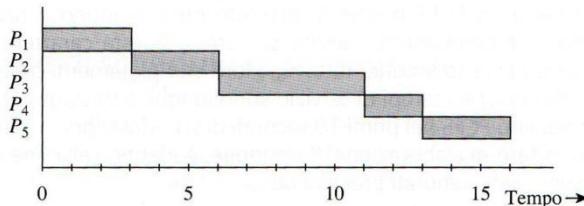
ESEMPIO:

Ricordiamo che:

- il tempo di ammissione di un processo è il momento in cui il processo viene sottomesso al sistema
- il tempo di servizio di un processo è il tempo richiesto per completare la sua esecuzione
- il tempo di turnaround (ta) di un processo è il tempo impiegato per il completamento
- il completamento pesato (w) mette in relazione il tempo di completamento di un processo con il suo tempo di servizio

Processo	P_1	P_2	P_3	P_4	P_5
Tempo di ammissione	0	2	3	4	8
Tempo di servizio	3	3	5	2	3

Tempo	Processo completato			Processi nel sistema (in ordine FCFS)	Processo schedulato
	<i>id</i>	<i>ta</i>	<i>w</i>		
0	—	—	—	P_1	P_1
3	P_1	3	1.00	P_2, P_3	P_2
6	P_2	4	1.33	P_3, P_4	P_3
11	P_3	8	1.60	P_4, P_5	P_4
13	P_4	9	4.50	P_5	P_5
16	P_5	8	2.67	—	—



Come si può vedere, l'ordine di arrivo dei processi è $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$ che vengono eseguiti proprio nell'ordine con cui sono arrivati.

Il processo P_1 arriva al tempo 0 e viene subito schedulato in quanto è il primo (e l'unico) processo presente nella coda FIFO. Rimane in esecuzione per 3 unità di tempo per poi rilasciare la CPU.

Nel frattempo sono arrivati, nell'ordine, anche i processi P_2 e P_3 .

Il processo P_2 arriva al tempo 2, ma va in esecuzione non appena P_1 termina, cioè al tempo 3, quindi rimane in attesa per una unità di tempo. Al tempo 3 inizia la sua esecuzione che termina dopo 3 unità di tempo, cioè al tempo 6. Il suo tempo di turnaround è pari a 4 unità di tempo, in quanto la sua attesa è durata una unità di tempo mentre la sua esecuzione tre unità di tempo, quindi $1+3=4$.

Nel frattempo, al già presente processo P_3 , si è aggiunto nella coda il processo P_4 .

Il processo P_3 arriva al tempo 3, ma va in esecuzione non appena P_2 termina, cioè al tempo 6, quindi rimane in attesa per tre unità di tempo. Al tempo 6 inizia la sua esecuzione che termina dopo 5 unità di tempo, cioè al tempo 11. Il suo tempo di turnaround è pari a 8 unità di tempo, di cui 3 per l'attesa e 5 per l'esecuzione.

Nel frattempo, al già presente processo P_4 , si è aggiunto nella coda anche il processo P_5 .

Il processo P_4 arriva al tempo 4, ma va in esecuzione non appena P_3 termina, cioè al tempo 11, quindi rimane in attesa per sette unità di tempo. Al tempo 11 inizia la sua esecuzione che termina dopo 2 unità di tempo, cioè al tempo 13. Il suo tempo di turnaround è pari a 9 unità di tempo, di cui 7 riguardano l'attesa e 2 riguardano l'esecuzione.

Nella coda è presente solo il processo P_5 .

Il processo P_5 arriva al tempo 8, ma va in esecuzione non appena P_4 termina, cioè al tempo 13, quindi rimane in attesa per cinque unità di tempo. Al tempo 13 inizia la sua esecuzione che termina dopo 3 unità di tempo, cioè al tempo 16. Il suo tempo di turnaround è pari a 8 unità di tempo, di cui 5 riguardano l'attesa e 3 riguardano l'esecuzione.

6.3.2 Scheduling Shortest Job First (SJF) – Shortest Next Process First (SNPF)

Gli algoritmi SJF possono essere sia non preemptive (SNPF) che preemptive (SRTF). Qui tratteremo lo SNPF.

TIPO: lo scheduling SNPF è un algoritmo *senza prelazione*, con o senza priorità e dinamico.

FUNZIONAMENTO: seleziona il processo in attesa che userà la CPU per minor tempo. Se due processi hanno lo stesso tempo di esecuzione, verrà applicato lo scheduling FCFS. In pratica, le richieste brevi tendono a ricevere prima l'uso della CPU. In pratica, è come se il CPU-BURST fosse la priorità.

CONCLUSIONI: l'algoritmo SJF eleva il throughput (numero di processi portati a termine in un dato tempo) Presenta, però, due problematiche. Questo algoritmo privilegia i processi I/O bound.

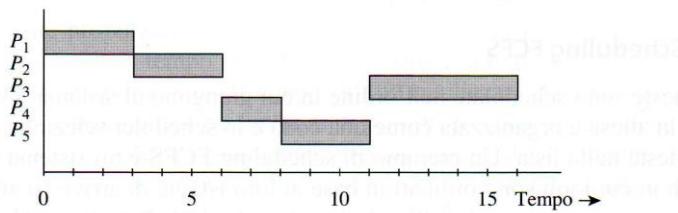
La prima è che è necessario conoscere in anticipo i tempi di esecuzione (tempo di servizio) dei vari processi. Visto che il SO non conosce a priori i tempi di servizio, occorre effettuare una stima dei CPU burst. Questa può essere effettuata conoscendo la 'storia passata' in modo tale da poter fare una stima del 'futuro'. Occorre comunque sapere che la media esponenziale si adatta meglio della media aritmetica per effettuare la stima.

La seconda è che possiede un potenziale problema di *starvation*, in cui è possibile che un processo rimanga in attesa troppo tempo prima di essere completato se vengono aggiunti continuamente piccoli processi alla coda dei processi pronti.

ESEMPIO:

Processo	P_1	P_2	P_3	P_4	P_5
Tempo di ammissione	0	2	3	4	8
Tempo di servizio	3	3	5	2	3

Tempo	Processo completato			Processi nel sistema (in ordine FCFS)	Processo schedulato
	<i>id</i>	<i>ta</i>	<i>w</i>		
0	–	–	–	{ P_1 }	P_1
3	P_1	3	1.00	{ P_2, P_3 }	P_2
6	P_2	4	1.33	{ P_3, P_4 }	P_4
8	P_4	4	2.00	{ P_3, P_5 }	P_5
11	P_5	3	1.00	{ P_3 }	P_3
16	P_3	13	2.60	{}	–



Come si può vedere, l'ordine di arrivo e quello di esecuzione possono non essere uguali.

Il processo P_1 arriva al tempo 0 e viene subito schedulato in quanto è il primo (e l'unico) processo presente nella coda. Rimane in esecuzione per 3 unità di tempo per poi rilasciare la CPU.

Quando termina, nella coda sono presenti i processi P_2 e P_3 . Il prossimo processo ad essere schedulato è il processo P_2 perché ha un tempo di servizio minore rispetto a P_3 .

Il processo P_2 arriva al tempo 2, ma va in esecuzione al tempo 3, quindi rimane in attesa per una unità di tempo. Al tempo 3 inizia la sua esecuzione che termina dopo 3 unità di tempo, cioè al tempo 6. Il suo tempo di turnaround è pari a 4 unità di tempo (1 di attesa + 3 di esecuzione = 4).

Quando termina, nella coda sono presenti i processi P_3 e P_4 . Il processo P_4 è quello col tempo di servizio minore, quindi sarà il prossimo ad essere schedulato.

Il processo P_4 arriva al tempo 4, ma va in esecuzione al tempo 6, quindi rimane in attesa per due unità di tempo. Al tempo 6 inizia la sua esecuzione che termina dopo 2 unità di tempo, cioè al tempo 8. Il suo tempo di turnaround è pari a 4 unità di tempo (2 di attesa + 2 di esecuzione = 4).

Quando termina, nella coda sono presenti i processi P_3 e P_5 . Quest'ultimo sarà il prossimo ad essere schedulato in quanto ha un tempo di servizio minore.

Il processo P_5 arriva al tempo 8 e va subito in esecuzione, quindi la sua attesa è nulla. Al tempo 8 inizia la sua esecuzione che termina dopo 3 unità di tempo, cioè al tempo 11. Il suo tempo di turnaround è pari a 3 unità di tempo (0 di attesa + 3 di esecuzione = 3).

Quando termina, nella coda è presente solo il processo P_3 che sarà schedulato.

Il processo P_3 arriva al tempo 3, ma va in esecuzione al tempo 11, quindi rimane in attesa per otto unità di tempo. Al tempo 11 inizia la sua esecuzione che termina dopo 5 unità di tempo, cioè al tempo 16. Il suo tempo di turnaround è pari a 13 unità di tempo (8 di attesa + 5 di esecuzione = 13).

6.3.3 Scheduling Highest Response Ratio Next (HRRN)

TIPO: lo scheduling HRRN è un algoritmo *senza prelazione*, con priorità e dinamico.

FUNZIONAMENTO: questo algoritmo viene utilizzato per prevenire l'*aging*, ossia l'attesa eccessiva dei processi molto lunghi scavalcati da quelli più brevi, che avviene negli algoritmi SJF. Questa politica calcola i rapporti di risposta di tutti i processi nel sistema secondo la formula:

$$\text{Rapporto di risposta} = \frac{W+S}{S} \quad \text{dove } W \text{ è il tempo di attesa ed } S \text{ è il tempo di servizio.}$$

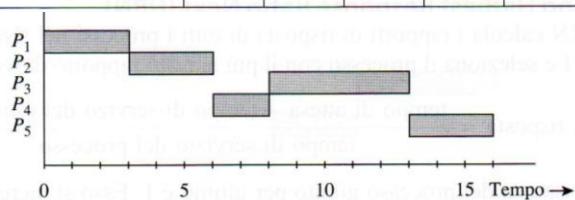
Questa politica schedula il processo con il rapporto di risposta maggiore.

CONCLUSIONI: l'obiettivo è quello di dare la possibilità ai processi con un CPU-BURST elevato di essere schedulati prima dei processi con CPU-BURST più piccolo.

Il rapporto di risposta dei processi brevi si incrementa più rapidamente di quello di un processo lungo; ma, comunque, il rapporto di risposta di un processo lungo, col passare del tempo, può diventare sufficientemente grande da consentire al processo di essere schedulato. Mediante questa politica si evita la *starvation* descritto nell'algoritmo precedente (SJF).

ESEMPIO:

Processo		P_1	P_2	P_3	P_4	P_5		
Tempo di ammissione		0	2	3	4	8		
Tempo di servizio		3	3	5	2	3		
Tempo	Processo completato						Processo schedulato	
	<i>id</i>	<i>ta</i>	<i>w</i>	P_1	P_2	P_3	P_4	P_5
0	—	—	—	1.00				P_1
3	P_1	3	1.00		1.33			P_2
6	P_2	4	1.33			1.60		P_4
8	P_4	4	2.00			2.00		P_3
13	P_3	10	2.00				1.00	P_5
16	P_5	8	2.67				2.67	—



Come si può vedere, anche in questa politica, l'ordine di arrivo e di esecuzione dei processi non è lo stesso.

Per la schedulazione occorre prendere in considerazione il rapporto di risposta (*response ratio*) dei processi.

Il processo P_1 arriva al tempo 0 e viene subito schedulato in quanto è il primo (e l'unico) processo presente nella coda. Rimane in esecuzione per 3 unità di tempo per poi rilasciare la CPU.

Quando termina, nella coda sono presenti i processi P_2 e P_3 . Il prossimo processo ad essere schedulato è il processo P_2 perché ha un rapporto di risposta maggiore rispetto a P_3 .

Il processo P_2 arriva al tempo 2, ma va in esecuzione al tempo 3, quindi rimane in attesa per una unità di tempo. Al tempo 3 inizia la sua esecuzione che termina dopo 3 unità di tempo, cioè al tempo 6. Il suo tempo di turnaround è pari a 4 unità di tempo (1 di attesa + 3 di esecuzione = 4).

Quando termina, nella coda sono presenti i processi P_3 e P_4 . Il processo P_4 è quello col rapporto di risposta maggiore, quindi sarà il prossimo ad essere schedulato.

Il processo P_4 arriva al tempo 4, ma va in esecuzione al tempo 6, quindi rimane in attesa per due unità di tempo. Al tempo 6 inizia la sua esecuzione che termina dopo 2 unità di tempo, cioè al tempo 8. Il suo tempo di turnaround è pari a 4 unità di tempo (2 di attesa + 2 di esecuzione = 4).

Quando termina, nella coda sono presenti i processi P_3 e P_5 . Il primo sarà il prossimo ad essere schedulato in quanto ha un rapporto di risposta maggiore.

Il processo P_3 arriva al tempo 3, ma va in esecuzione al tempo 8, quindi rimane in attesa per cinque unità di tempo. Al tempo 8 inizia la sua esecuzione che termina dopo 5 unità di tempo, cioè al tempo 13. Il suo tempo di turnaround è pari a 10 unità di tempo (5 di attesa + 5 di esecuzione = 10).

Quando termina, nella coda è presente solo il processo P_5 che sarà schedulato.

Il processo P_5 arriva al tempo 8, ma va in esecuzione al tempo 13, quindi rimane in attesa per cinque unità di tempo. Al tempo 13 inizia la sua esecuzione che termina dopo 3 unità di tempo, cioè al tempo 16. Il suo tempo di turnaround è pari a 8 unità di tempo (5 di attesa + 3 di esecuzione = 3).

6.4 Politiche di scheduling preemptive

Nello *scheduling preemptive* i processi possono essere prelazionati, in favore di altri processi, prima che possano completare le proprie operazioni. Il processo interrotto viene riportato all'interno della lista dei processi *ready* in modo tale che prima o poi possa essere rielezionato dallo scheduler. Per queste ragioni, una richiesta può essere schedulata anche diverse volte prima che venga completata. Questa caratteristica genera un maggior overhead rispetto allo scheduling non preemptive.

Le tre politiche di scheduling preemptive sono:

- Scheduling *Shortest Job First* (SJF) nella versione preemptive (SRTF)
- Scheduling *Round-Robin con Time-Slicing* (RR)
- Scheduling *Least Completed Next* (LCN)

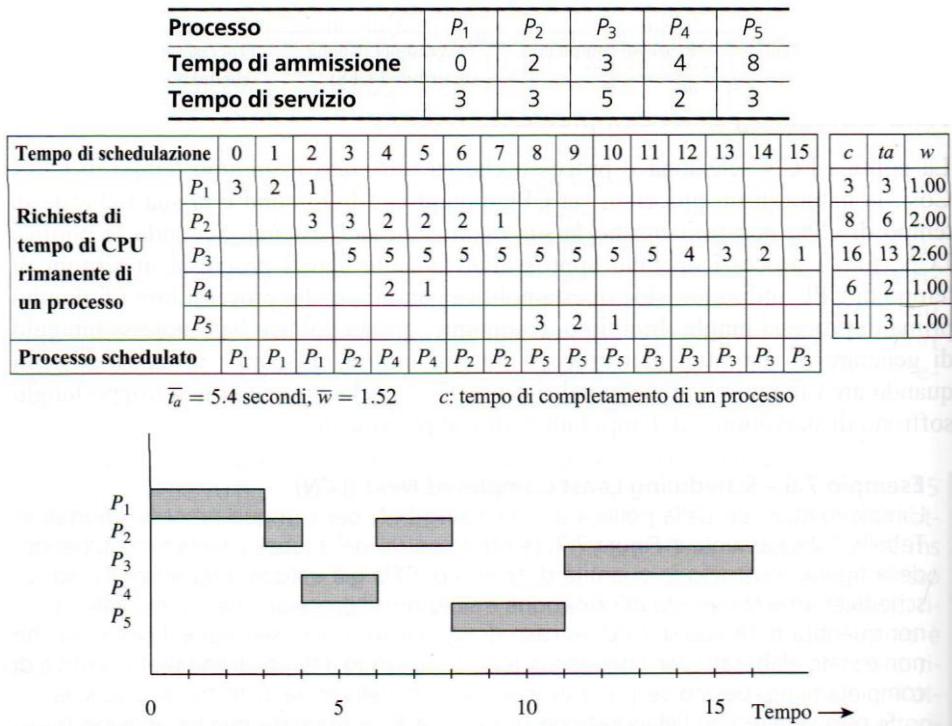
6.4.1 Scheduling Shortest Job First (SJF) – **Shortest Remaining Time First (SRTF)**

Lo scheduling SRTF non è altro che la versione preemptive (con prelazione) dello scheduling SJF.

FUNZIONAMENTO: si differenzia dallo SNPF per il fatto che, quando viene sottomesso un nuovo processo la cui durata è minore del tempo necessario al processo in esecuzione per concludere le proprie operazioni, lo scheduler provvede ad effettuare un context switch per assegnare l'uso della CPU al nuovo processo. In caso di uguaglianza viene selezionato il processo che non è stato elaborato per il periodo di tempo più lungo.

CONCLUSIONI: un nuovo processo con CPU burst minore del tempo rimasto all'attuale processo in esecuzione prelaziona quest'ultimo. In pratica favorisce i processi più brevi rispetto a quelli più lunghi. Poiché è analoga alla politica SNPF, i processi lunghi possono andare incontro a starvation.

ESEMPIO:



6.4.2 Scheduling Round-Robin con Time-Slicing (RR)

TIPO: lo scheduling RR è un algoritmo *con prelazione*, con o senza priorità e statico o dinamico.

FUNZIONAMENTO: ad ogni processo viene assegnato un intervallo di tempo, chiamato *quanto* (time slice), durante il quale al processo è assegnato l'uso della CPU. Per scandire i quanti, alla fine di ognuno di essi viene generato un timer interrupt.

Se alla fine del quanto, il processo non ha terminato le sue operazioni, allora viene prelazionato e inserito di nuovo nella coda, e la CPU viene assegnata ad un altro processo.

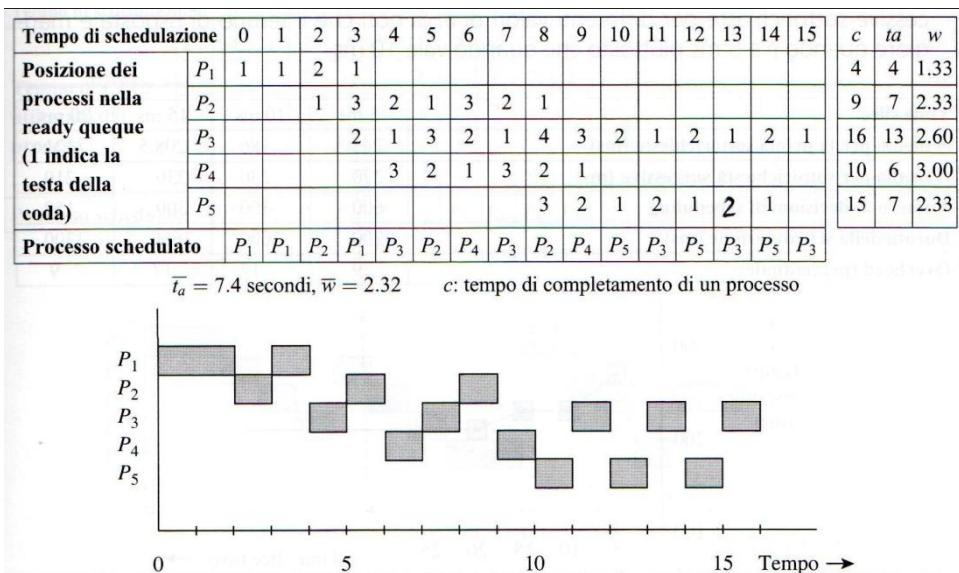
Se prima della fine del quanto, il processo si blocca o termina le sue operazioni, allora viene selezionato un altro processo a cui assegnare la CPU.

CONCLUSIONI: questo algoritmo privilegia i processi CPU bound perché utilizza l'intero quanto assegnato. Il round robin è facile da implementare: lo scheduler mantiene una coda di processi in stato *ready* e seleziona semplicemente il primo processo in coda e quando scade il quanto il processo viene messo in fondo alla lista. L'unica questione riguardo a questo algoritmo è la **durata del quanto**: assegnare un quanto troppo breve provoca troppi context switch e peggiora l'efficienza della CPU (overhead troppo elevato), ma assegnarlo troppo alto può provocare tempi di risposta lunghi per richieste interattive brevi.

Se nella coda dei processi pronti esistono n processi e il quanto di tempo è pari a q , ciascun processo ottiene un $1/n$ -esimo del tempo di elaborazione della CPU, in frazioni di, al massimo, q unità di tempo.

ESEMPIO:

Processo	P_1	P_2	P_3	P_4	P_5
Tempo di ammissione	0	2	3	4	8
Tempo di servizio	3	3	5	2	3



Come si può vedere, anche qui, l'ordine di arrivo e di esecuzione dei processi può non essere lo stesso.

Nella tabella, la prima riga indica gli istanti in cui vengono prese le decisioni.

Dalla seconda alla sesta riga vengono mostrate le posizioni dei cinque processi nella coda. Una casella bianca indica che il processo non è nel sistema in quell'istante.

L'ultima riga mostra il processo selezionato dallo scheduler, cioè il processo a cui viene attribuita la CPU.

Nella tabella a destra sono riportati il tempo di **completamento** (c), il tempo di **turnaround** (ta) e il **completamento pesato** (w).



In questo esempio si assume che un nuovo processo che è ammesso nel sistema nello stesso istante in cui un processo è sospeso (cioè si blocca) sarà inserito nella coda prima del processo sospeso. Inoltre, si assume che il quanto duri 1 secondo dunque lo scheduler prende decisioni di scheduling ogni secondo.

Al tempo 0 e al tempo 1, nella coda è presente solo P_1 , quindi viene mandato in esecuzione.

Al tempo 2 arriva il processo P_2 , la coda contiene i processi nell'ordine $P_1 \rightarrow P_2$, quindi esegue P_2 .

Al tempo 3 arriva il processo P_3 , la coda contiene i processi nell'ordine $P_1 \rightarrow P_3 \rightarrow P_2$, quindi esegue P_1 che conclude le sue operazioni e lascia la coda.

Al tempo 4 arriva il processo P_4 , la coda contiene i processi nell'ordine $P_3 \rightarrow P_2 \rightarrow P_4$, quindi esegue P_3 .

Al tempo 5, l'ordine è $P_2 \rightarrow P_4 \rightarrow P_3$, quindi viene eseguito P_2 .

...

Al tempo 8 arriva il processo P_5 , nella coda l'ordine è $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_3$, quindi esegue P_2 che conclude le sue operazioni e lascia la coda.

Al tempo 9, l'ordine è $P_4 \rightarrow P_5 \rightarrow P_3$, viene eseguito P_4 che conclude le sue operazioni e lascia la coda.

...

Al tempo 14, l'ordine è $P_5 \rightarrow P_3$, viene eseguito P_5 che conclude le sue operazioni e lascia la coda.

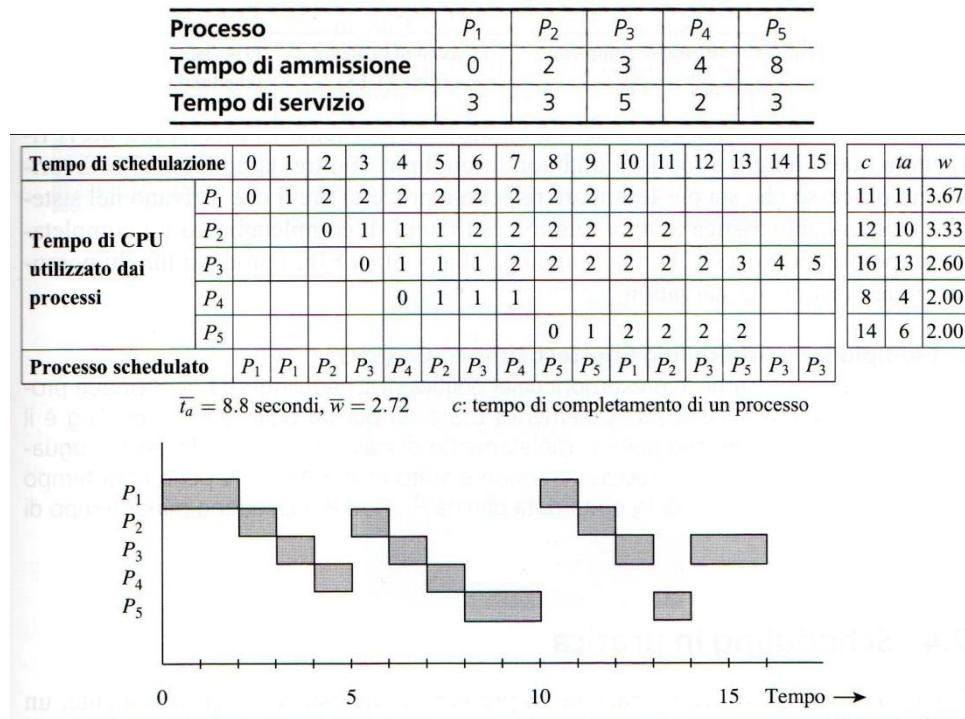
Al tempo 15 è presente solo P_3 che viene eseguito e conclude le sue operazioni.

6.4.3 Scheduling Least Completed Next (LCN)

FUNZIONAMENTO: la politica LCN seleziona il processo che ha utilizzato il minimo tempo di CPU. In caso di uguaglianza lo scheduler seleziona il processo che non è stato elaborato per il periodo di tempo più lungo.

CONCLUSIONI: tutti i processi operano approssimativamente eguali progressi in termini di tempo di CPU utilizzato, cioè questa politica garantisce che processi brevi finiranno prima di processi lunghi (I/O bound).

Ha il noto **svantaggio** di generare la **starvation** dei processi lunghi. Inoltre, trascura i processi esistenti quando arrivano nuovi processi nel sistema. Per questo motivo anche i processi non troppo lunghi soffrono di starvation o di tempi lunghi di completamento.

**ESEMPIO:**

Come si può vedere, anche qui, l'ordine di arrivo e di esecuzione dei processi può non essere lo stesso.

Nella tabella, la prima riga indica gli istanti in cui vengono prese le decisioni.

Dalla seconda alla sesta riga vengono mostrate le quantità di tempo di CPU già utilizzate da ciascun processo. Una casella bianca indica che il processo non è nel sistema in quell'istante.

L'ultima riga mostra il processo selezionato dallo scheduler, cioè il processo a cui viene attribuita la CPU.

Nella tabella a destra sono riportati il tempo di completamento (c), il tempo di turnaround (ta) e il completamento pesato (w).

Al tempo 0 e al tempo 1 viene eseguito P_1 perché è il solo processo a trovarsi in memoria.

Al tempo 2 arriva il processo P_2 , viene eseguito P_2 .

Al tempo 3 arriva il processo P_3 , viene eseguito P_3 .

Al tempo 4 arriva il processo P_4 , viene eseguito P_4 .

Al tempo 5, i processi P_2 , P_3 e P_4 hanno lo stesso tempo di CPU utilizzato, ma viene eseguito P_2 perché è quello che sta in attesa da più tempo.

Al tempo 6, i processi P_3 e P_4 hanno lo stesso tempo di CPU utilizzato, ma viene eseguito P_3 perché è quello che sta in attesa da più tempo.

Al tempo 7 viene eseguito P_4 perché è l'unico che ha usato per minor tempo la CPU e conclude le sue operazioni lasciando la coda.

Al tempo 8 arriva il processo P_5 , viene eseguito P_5 .

Al tempo 9 viene eseguito P_5 perché è l'unico che ha usato per minor tempo la CPU.

Al tempo 10, i processi P_1 , P_2 , P_3 e P_5 hanno lo stesso tempo di CPU utilizzato, ma viene eseguito P_1 perché è quello che sta in attesa da più tempo. P_1 termina le proprie operazioni.

Al tempo 11, i processi P_2 , P_3 e P_5 hanno lo stesso tempo di CPU utilizzato, ma viene eseguito P_2 perché è quello che sta in attesa da più tempo. P_2 termina le proprie operazioni.

Al tempo 12, i processi P_3 e P_5 hanno lo stesso tempo di CPU utilizzato, ma viene eseguito P_3 perché è quello che sta in attesa da più tempo.

Al tempo 13 viene eseguito P_5 che termina le proprie operazioni lasciando la coda.

Al tempo 14 e al tempo 15 è presente solo P_3 che conclude le sue operazioni.

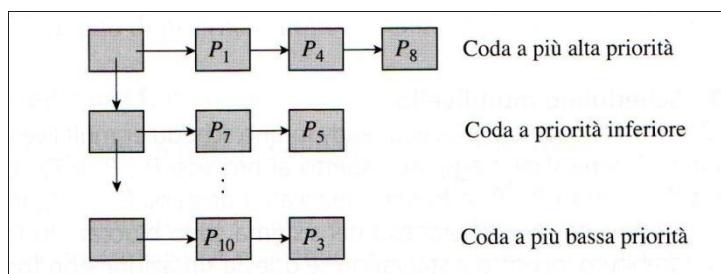
6.5 Scheduling in pratica

6.5.1 Meccanismi di scheduling

Cosa dovrebbe fare lo scheduler se non ci sono processi *ready*? In tal caso, non c'è "lavoro" per la CPU; comunque la CPU deve essere pronta a gestire qualsiasi interrupt che uno dei processi *blocked* può attivare. A tal fine, tipicamente, un kernel esegue un *idle loop*, cioè un loop senza fine che non contiene istruzioni. Questa soluzione consuma energia, quindi nei sistemi mobili o embedded, dove il consumo è una delle priorità, è essenziale evitare questo spreco di energia.

Per rispondere a questo requisito, i computer sono dotati di una modalità speciale, chiamata *sleep mode*, in cui la CPU non esegue istruzioni ma è in grado di accettare gli interrupt. Alcuni computer, addirittura, oltre a non eseguire istruzioni, calano anche il clock della CPU e disconnettono quest'ultima dal bus di sistema.

6.5.2 Scheduling basato su priorità



La figura mostra una soluzione efficiente per le politiche di scheduling basate su priorità.

La schedulazione round-robin ipotizza che tutti i processi abbiano la stessa importanza. La necessità di prendere in considerazione fattori esterni porta alla schedulazione con priorità: in pratica viene mantenuta una lista separata di processi *ready* per ogni valore di priorità; questa lista è organizzata come una coda di PCB, ognuno dei quali punta al PCB del successivo processo in coda.

La testa della coda contiene due puntatori: uno che punta al PCB del primo processo nella coda, l'altro che punta alla testa della coda con priorità inferiore.

Lo scheduler parte dalla testa della coda con priorità maggiore e man mano scorre le teste delle code con ordine decrescente di priorità. Seleziona il primo processo nella prima coda non vuota che trova.

Le priorità possono essere assegnate *dinamicamente* o *staticamente*. Lo scheduling basato su priorità può condurre alla starvation dei processi a bassa priorità. Mediante l'*aging* potrebbe essere risolto questo problema. Se si utilizza la tecnica dell'*aging*, le priorità devono essere considerate *dinamiche*, cioè il PCB di un processo dovrebbe essere spostato tra diverse code dei processi.

6.5.3 Scheduling multilivello

Questo scheduling è detto anche *scheduling a code multiple*. Esso combina lo scheduling basato su priorità e quello round-robin per fornire una buona combinazione tra prestazione del sistema e tempi di risposta.

Uno scheduler multilivello utilizza varie code di processi pronti, e ad ogni classe è associata una priorità. Alla coda con priorità più alta viene assegnato un *quanto* più piccolo, mentre alle code con priorità via via minore viene assegnato un *quanto* sempre più grande. La coda con priorità più bassa avrà il *quanto* più grande.

TIPO: lo scheduling multilivello è un algoritmo *con prelazione*, con priorità e statico.

FUNZIONAMENTO: abbiamo detto che vengono definite classi di priorità. I processi della classe più alta vengono eseguiti per un quanto, quelli della classe successiva per due quanti, quelli della classe seguente per quattro quanti e così via. Ogni qualvolta che un processo rimane in esecuzione per tutto il quanto, viene abbassato di una classe. Se, per esempio, un processo A ha bisogno di 100 quanti, inizialmente gli viene assegnato un quanto, dopodiché viene schedulato un altro processo. Quando A riottiene la CPU, gli vengono assegnati due quanti, poi 4, poi 8, poi 16, poi 32, poi 64 quanti. In pratica ottiene la CPU per sette volte anziché per cento come sarebbe avvenuto con il round-robin.

Un processo in testa a una coda è selezionato dallo scheduler solo se le code con priorità più alta sono vuote. Questo tipo di scheduling è di tipo *preemptive*, quindi un processo può essere prelazionato se arriva un processo con priorità più alta; in questo caso, il processo interrotto viene aggiunto alla fine della coda a cui apparteneva.

Per ottenere un'efficienza ottimale, si dà una priorità elevata ai processi I/O bound, mentre si dà una priorità bassa ai processi CPU bound.

In questa politica di scheduling vengono utilizzate *priorità statiche*, per questo motivo non può prevenire la *starvation* dei processi a bassi livelli di priorità.

6.5.4 Scheduling multilivello adattivo

Questo scheduling è anche detto *scheduling a code multiple con feedback*.

Ciò che lo contraddistingue dallo scheduling multilivello è l'uso delle priorità dinamiche, ciò permette allo scheduling di variare la priorità di un processo in modo tale da evitare la starvation dei processi. Lo scheduler analizza il comportamento passato di un processo per modificare, eventualmente, la sua priorità. Quindi, l'uso delle priorità dinamiche consente ad un processo di muoversi tra le varie code presenti.

6.5.5 Scheduling fair share

Tutte le politiche di scheduling viste finora si occupano di fornire servizi equi ai processi piuttosto che agli utenti o alle loro applicazioni. Se le applicazioni creano un diverso numero di processi, un'applicazione che impiega più processi è probabile che riceva maggiore attenzione dalla CPU rispetto ad un'applicazione che utilizza meno processi.

La nozione di *fair share* risponde proprio a questa esigenza. Una fair share assicura un equo utilizzo della CPU da parte degli utenti o delle applicazioni. In pratica, gli utenti o le applicazioni che usano un numero più alto di processi ricevono meno risorse.

SCEDULING A LOTTERIA

Lo scheduling a lotteria è una tecnica utile per la condivisione di una risorsa in maniera probabilisticamente equa. I '*biglietti*' sono distribuiti a tutti i processi che condividono una risorsa (ad esempio il tempo di CPU).

Quando lo scheduler deve selezionare un processo a cui attribuire la CPU, viene scelto a caso un biglietto e il processo che lo possiede ottiene la risorsa. Per migliorare le prestazioni, ai processi più importanti possono essere assegnati biglietti extra, per aumentare la loro probabilità di vincere.

Inoltre i processi possono scambiarsi i biglietti se lo desiderano per avvantaggiare i processi con la quale cooperano per raggiungere un obiettivo comune.

6.6 Scheduling real-time

Nello scheduling real-time il tempo gioca un ruolo essenziale perché si devono rispettare delle scadenze chiamate *deadline*.

I Sistemi real-time sono classificati in:

- **Hard real-time**: garantiscono che i compiti critici sono completati in un intervallo limitato; cioè rispetta le deadline in maniera garantita.
- **Soft real-time**: sono meno restrittivi; cioè che pur avendo scadenze da rispettare, se alcuni processi non lo fanno occasionalmente, sono tollerati; cioè rispetta le deadline in maniera probabilistica.

Gli eventi a cui un sistema real-time può dover reagire possono essere ulteriormente classificati in:

- **periodici**: che si verificano ad intervalli regolari
- **aperiodici**: che si verificano in modo imprevedibile

Gli algoritmi di schedulazione real-time possono essere inoltre:

- **statici**: cioè che prendono le decisioni di scheduling prima che il sistema inizi l'esecuzione. La schedulazione statica funziona solo quando sono disponibili in anticipo le informazioni complete circa il lavoro da fare e le scadenze da rispettare;
- **dinamici**: cioè che le prendono durante l'esecuzione. Mentre gli algoritmi di scheduling dinamica non hanno queste restrizioni.

6.6.1 Precedenze e schedulabilità di un processo

I processi di un'applicazione real-time interagiscono tra loro per assicurare che le loro azioni vengono eseguite nel giusto ordine. Per gestire le dipendenze tra i processi viene usato un *grafo di precedenza tra processi*.

Abbiamo detto che i sistemi real-time possono essere sia *soft* che *hard*. La nozione di *schedulabilità* aiuta a differenziare queste situazioni. La **schedulabilità** è una sequenza di decisioni di scheduling che consentono ai processi di un'applicazione di agire secondo le precedenze e rispettare il requisito di risposta dell'applicazione.

Approccio	Descrizione
Scheduling statico	Uno schedule viene preparato <i>prima</i> che l'esecuzione dell'applicazione real-time cominci, tenendo conto delle interazioni tra processi, le periodicità, i vincoli sulle risorse e le scadenze.
Scheduling basato su priorità	L'applicazione real-time viene analizzata per attribuire appropriate <i>priorità</i> ai suoi processi. Durante l'esecuzione dell'applicazione viene adottato lo scheduling convenzionale basato su priorità.
Scheduling dinamico	Lo scheduling è eseguito quando proviene una richiesta di creazione di un processo. La creazione di un processo avviene solo se il requisito di risposta del processo può essere soddisfatto in maniera garantita.

La tabella riassume i tre approcci principali allo scheduling real-time.

6.6.2 Scheduling della deadline

Per ogni processo possono essere specificati due tipi di deadline:

- la **scadenza d'inizio**, cioè il minimo istante entro cui le operazioni del processo devono cominciare
- la **scadenza di fine**, cioè il tempo entro il quale le operazioni del processo devono terminare

Esistono vari modi per determinare le scadenze che prendono in considerazione diversi fattori: le precedenze del processo, la possibilità di eseguire operazioni di I/O per i processi, la disponibilità delle risorse, ecc.

6.7 Casi di studio

6.7.1 Scheduling in Unix

Unix è un SO time-sharing puro che usa una politica di scheduling multilivello adattivo, quindi viene associata ad ogni processo una priorità, che in Unix è di tipo numerica: valori alti corrispondono a basse priorità e viceversa. Vengono schedulati i processi con priorità maggiore (cioè col valore vicino allo 0).

Ad esempio, in Unix 4.3, le priorità variano nell'intervallo 0-127. I processi in modalità utente hanno una priorità tra 50 e 127, mentre quelli in modalità kernel hanno priorità tra 0 e 49.

In Unix, viene usata la seguente formula per variare la priorità di un processo:

$$\text{Priorità processo} = \text{priorità base per processi utente} + f + \text{valore nice}$$

dove:

- f è il tempo di CPU usato recentemente
- nice è una system call che permette di modificare la priorità del processo chiamante

Con l'uso del secondo fattore dell'equazione, cioè f , viene assicurata un'equa divisione del tempo di CPU tra i gruppi di processi (*scheduling fair share*). Infatti, la f è relativa all'uso della CPU, più è alto il suo valore, minore sarà la priorità del processo. Quindi se un processo ha usato la CPU recentemente, vedrà la sua priorità diminuire.

La system call *nice* permette di modificare la priorità del processo chiamante. Visto che può accettare solo un valore ≥ 0 , il terzo parametro dell'equazione non può far altro che diminuire la priorità del processo. Un processo potrebbe invocare una *nice* per favorire altri processi con i quali coopera per il raggiungimento di un obiettivo comune.

6.7.2 Scheduling in Linux

Linux supporta sia applicazioni real-time che applicazioni non real-time.

I *processi real-time* possiedono priorità *statiche* tra 0 e 100, dove 0 è la priorità più alta, e possono essere schedulati in due modi: FIFO o round-robin. Un flag associato ad ogni processo indica il modo in cui il processo deve essere schedulato.

I *processi non real-time* possiedono priorità più basse di quelli real-time; sono priorità *dinamiche* che variano tra -20 e 19, dove -20 è la priorità più alta.

Quindi il kernel ha (100+40) livelli di priorità.

Anche in Linux, è possibile modificare la priorità di un processo non real-time mediante la system call *nice*, ma sono comunque necessari dei privilegi speciali per richiamarla. Ovviamente, questa system call non fa altro che diminuire la priorità di un processo, quando quest'ultimo vuole "favorire" un altro processo (ad esempio perché stanno collaborando per la risoluzione di un obiettivo comune).

I processi non real-time vengono schedulati usando la nozione di *time-slice*. Quando un processo esaurisce la sua time-slice (*quanto*) senza completare le sue operazioni, andrà dinuovo in esecuzione solo dopo che tutti gli altri processi abbiano esaurito la loro time-slice.

Linux utilizza le time-slice nell'intervallo compreso tra 10 e 200ms. Per assicurare che un processo a più alta priorità riceva maggiore tempo di CPU rispetto ad un processo a priorità inferiore, Linux assegna una time-slice maggiore al processo a priorità più alta. Ciò non influenza i tempi di risposta in quanto un processo ad alta priorità sarebbe di natura interattiva, quindi eseguirebbe un'operazione di I/O prima di usare abbastanza tempo di CPU.

CAPITOLO 7. Sincronizzazione dei processi: memoria condivisa

I processi di un'applicazione cooperano per il raggiungimento di un obiettivo comune condividendo dati e coordinandosi tra loro. Questi processi possono essere chiamati *processi intercomunicanti*.

I concetti chiave della *sincronizzazione dei processi* sono essenzialmente due: la *sincronizzazione per l'accesso ai dati* e la *sincronizzazione per il controllo*. La prima riguarda l'uso della *mutua esclusione* per salvaguardare la consistenza dei dati condivisi, mentre la seconda riguarda l'uso delle *operazioni atomiche* per il coordinamento delle attività dei processi.

In questo capitolo verranno illustrate le *sezioni critiche*, ovvero porzioni di codice in cui si accede a dati condivisi in maniera mutuamente esclusiva, le *operazioni di invio dei segnali* indivisibili, usate per implementare il controllo della sincronizzazione, e verrà mostrato come entrambe sono implementate utilizzando operazioni indivisibili (dette anche atomiche).

Inoltre verranno illustrati alcuni problemi classici di sincronizzazione dei processi e come possono essere risolti usando tecniche (*semafori* e *monitor*) messe a disposizione dai linguaggi di programmazione e dai SO.

7.1 Cos'è la sincronizzazione dei processi?

In questo capitolo, utilizzeremo il termine *processo* come termine generico sia per i processi che per i thread.

Solitamente un'applicazione è composta da vari processi che interagiscono tra loro per il raggiungimento di un obiettivo comune. Questi processi sono detti *processi concorrenti* o *processi intercomunicanti*.

Due o più processi possono interagire fra loro secondo due modalità: *cooperazione* e *competizione*.

- due processi *cooperano* se ciascuno ha bisogno dell'altro per procedere con le proprie operazioni.
- due processi *competono* se entrano in conflitto sulla ripartizione delle risorse.

In entrambi i casi occorre predisporre meccanismi di sincronizzazione e comunicazione che permettano al processo di gestire la cooperazione e la competizione.

7.1.1 Processi concorrenti

Col termine *processi concorrenti* ci si riferisce a processi il cui comportamento è influenzato dalla contemporanea presenza di altri processi.

I *processi intercomunicanti* sono processi concorrenti che condividono dati o che coordinano le loro attività.

I processi che non interagiscono tra loro sono detti *processi indipendenti*.

La *sincronizzazione dei processi* riguarda i processi intercomunicanti e consiste nell'individuare le tecniche usate per ritardare e ripristinare i processi e per implementare le interazioni tra i processi stessi.

7.1.2 Interazione tra processi

Il modo in cui i processi interagiscono può essere classificato sulla base del grado di conoscenza che hanno dell'esistenza degli altri processi.

PROCESSI CHE NON SI VEDONO TRA LORO

Sono processi indipendenti che non sono fatti per collaborare. In questo caso il SO deve preoccuparsi della gestione della *competizione* per le risorse.

PROCESSI CHE VEDONO GLI ALTRI PROCESSI INDIRETTAMENTE

Sono processi che non conoscono necessariamente il nome degli altri processi, ma condividono con loro l'accesso a qualche oggetto, come un buffer di I/O. Tali processi effettuano *cooperazione* nel senso che condividono un oggetto comune.

PROCESSI CHE VEDONO GLI ALTRI PROCESSI DIRETTAMENTE

Sono processi che possono comunicare direttamente fra loro per nome, e che sono progettati per lavorare insieme; anche questi processi effettuano *cooperazione*.

7.2 Sincronizzazione per l'accesso ai dati e sezioni critiche

La *sincronizzazione per l'accesso ai dati* serve per garantire la consistenza dei dati condivisi. Si tratta, in pratica, di coordinare i processi per implementare la *mutua esclusione*.

In informatica il termine *mutex* (contrazione dell'inglese mutual exclusion, *mutua esclusione*) indica il meccanismo di sincronizzazione con il quale si impedisce che più processi accedano contemporaneamente agli stessi dati in memoria o ad altre risorse soggette a race condition.

La mutua esclusione viene implementata usando le *sezioni critiche* (CS): queste sono sezioni del codice che possono essere eseguite da un solo processo per volta. In pratica, se un processo P₁ sta eseguendo una sezione critica ed un processo P₂ vuole eseguire quella sezione critica, P₂ dovrà attendere finché P₁ non termini l'esecuzione della sua sezione critica.

In un codice possono esistere molte sezioni critiche.

7.3.1 Proprietà dell'implementazione di una sezione critica

Affinchè sia possibile la mutua esclusione occorrono tre condizioni:

1. *mutua esclusione*: un solo processo alla volta può accedere alla sezione critica
2. *progresso*: l'uso di una sezione critica non può essere "riservato" ad un processo; in pratica, nessun processo fuori dalla sezione critica può impedire ad un altro processo di entrare
3. *attesa limitata*: nessun processo deve attendere indefinitivamente prima di entrare nella sezione critica

Queste ultime due condizioni, assieme, garantiscono che non si verificherà mai il fenomeno della *starvation*.

7.3 Sincronizzazione per il controllo e operazioni indivisibili

Nel paragrafo precedente abbiamo discusso del primo concetto chiave della sincronizzazione dei processi, ovvero la *sincronizzazione per l'accesso ai dati*.

In questo paragrafo discuteremo del secondo concetto chiave, cioè la *sincronizzazione per il controllo*; essa serve per coordinare le attività dei processi in modo da eseguirli nell'ordine desiderato.

Questa sincronizzazione viene implementata mediante l'uso delle *operazioni atomiche*. Un'operazione atomica (detta anche *operazione indivisibile*) è il mezzo che assicura l'esecuzione di una sequenza di istruzioni senza essere prelazionati. Quindi, un'operazione atomica non può essere eseguita concorrentemente con ogni altra operazione che coinvolge gli stessi dati.

Un'operazione atomica su un insieme di dati è come una sezione critica. Tuttavia, facciamo distinzione tra i due termini perché una sezione critica deve essere implementata esplicitamente in un programma, mentre l'hardware o il software di un computer possono fornire alcune operazioni indivisibili tra le proprie operazioni primitive.

7.4 Race condition

Nell'ambito dei SO, le **race condition** (o *corse critiche*) sono delle situazioni che si verificano quando due o più processi stanno leggendo o scrivendo un qualche dato condiviso e il risultato finale dipende dall'ordine in cui vengono schedulati i processi. In pratica si ha una race condition quando due processi accedono alla stessa parte di memoria contemporaneamente.

Per evitare le *race condition* bisogna assicurarsi che solo un processo alla volta possa accedere e modifica i dati in comune. Questa condizione richiede una forma di sincronizzazione tra processi.

7.2.1 Condizioni di Bernstein



La programmazione concorrente risulta particolarmente efficiente in quanto permette di portare avanti più lavori simultaneamente avendo di conseguenza un tempo di risposta e di elaborazione migliore.

Dato un programma sequenziale è possibile stabilire un criterio per determinare se più istruzioni possono essere eseguite *concorrentemente*. Tale criterio è espresso dalle *condizioni di Bernstein*.

Mediante le condizioni di Bernstein è possibile prevenire le race condition, verificando per ogni coppia di processi se vengono soddisfatte le suddette condizioni.

Prima di elencare tali condizioni è necessario fornire alcune definizioni.

- indichiamo con A, B, C, ..., Y, Z le aree di memoria
- chiamiamo *dominio* (*i*) le aree di memoria da cui dipende un'istruzione *i*
- chiamiamo *range* (*i*) le aree di memoria modificate dall'istruzione *i*

E' lecito eseguire concorrentemente due istruzioni *i,y* se valgono le seguenti condizioni:

1. $range(i) \cap range(y) = \emptyset$ → agiscono su aree di memoria differenti
2. $range(i) \cap dominio(y) = \emptyset$ → l'area su cui agisce un'istruzione non modifica l'area da cui dipende l'altra
3. $dominio(i) \cap range(y) = \emptyset$ → viceversa della condizione 2

Quando un insieme di istruzioni rispetta le condizioni di Bernstein il loro esito complessivo sarà sempre lo stesso indipendentemente dall'ordine e dalla velocità di esecuzione, altrimenti in caso di violazione, gli errori dipenderanno dall'ordine di esecuzione e dalle velocità relative generando *race condition*.

7.5 Approcci alla sincronizzazione

Ci sono vari modi per soddisfare i requisiti della mutua esclusione:

- **Approcci software** (o **algoritmico**). Consistono in algoritmi che gestiscono la mutua esclusione senza alcun supporto da parte del sistema operativo o dal linguaggio di programmazione.
- **Approcci hardware**. Istruzioni macchina speciali e disabilitazione delle interruzioni, che hanno il vantaggio di ridurre l'overhead, ma non sono soddisfacenti.
- **Approcci sistema operativo/linguaggio di programmazione**. Consistono in costrutti, funzioni e strutture dati.

7.5.1 Ciclare – Approccio software

Un processo (P0 o P1) che vuole eseguire la sua sezione critica controlla la variabile globale condivisa *turno*; questa variabile indica quale processo può entrare nella sezione critica. Infatti se tale variabile è uguale ad uno degli identificatori dei processi (0 o 1), il processo corrente può entrare nella propria sezione critica; altrimenti, il processo resta in attesa, continuando a ciclare, che la variabile turno assuma valore uguale al proprio identificatore.

while (qualche processo è nella sezione critica su $\{d_s\}$ o
sta eseguendo un'operazione indivisibile utilizzando $\{d_s\}$)
{ non fare niente }

Sezione critica oppure
operazione indivisibile
tramite $\{d_s\}$

In pratica, nel ciclo while, il processo controlla se qualche processo è in sezione critica per lo stesso dato; in caso negativo, entra in sezione critica; in caso affermativo, continua a ciclare finché l'altro processo termina.

Quest'ultima situazione prende il nome di *busy waiting* poiché si mantiene la CPU durante l'esecuzione di un processo anche se il processo non fa nulla. Questa attesa, chiamata *attesa attiva*, termina solo quando il processo verifica che nessun altro processo è in una sezione critica.

7.5.2 Bloccare – Approccio hardware

Per evitare le attese attive, un processo in attesa di entrare in una sezione critica deve andare nello stato *blocked* invece di continuare a ciclare inutilmente. Lo stato del processo dovrebbe essere impostato a *ready* solo quando gli è consentito di entrare in sezione critica.

if (qualche processo è in una sezione critica su $\{d_s\}$ o
sta eseguendo un'operazione indivisibile usando $\{d_s\}$)
effettua una chiamata di sistema per bloccarsi;

Sezione critica oppure
operazione indivisibile
tramite $\{d_s\}$

Entrambe le soluzioni possono evitare race condition ma sono realizzate in due modi diversi.

Nel primo approccio (quello dove si cicla), detto *approccio algoritmico* (o *software*), viene adottata una complessa organizzazione di controlli per evitare race condition. Discuteremo di questo approccio nel paragrafo 7.7.

Il secondo approccio (quello dove si blocca) utilizza alcune caratteristiche dell'hardware per semplificare questo controllo. Discuteremo di questo approccio nel prossimo paragrafo.

7.6 Approccio hardware

Nell'approccio hardware, la sincronizzazione tra processi viene implementata utilizzando istruzioni macchina speciali fornite dall'architettura unite all'uso di variabili condivise, chiamate variabili di lock.

7.6.1 Disabilitazione delle interruzioni

Su un sistema monoprocesso, la soluzione più semplice è di permettere a ciascun processo di disabilitare le interruzioni non appena entra nella sua sezione critica in modo tale che il processo non possa essere prelazionato, e di riabilitarle non appena ne esce. Questo approccio ha però dei difetti: e se un processo disabilitasse le interruzioni e non le riabilitasse più? Potrebbe essere la fine del sistema.

Inoltre, questa soluzione non è attuabile su un sistema multiprocessore, cioè con due o più CPU, in quanto la disabilitazione delle interruzioni avrebbe effetto solo sul processore che esegue l'istruzione *disable*, mentre le altre CPU continuerebbero l'esecuzione e potrebbero accedere alla memoria condivisa.

Per questi motivi, i SO implementano le sezioni critiche e le operazioni atomiche attraverso *istruzioni indivisibili* fornite dai computer, insieme a variabili condivise chiamate *variabili di lock*.

VARIABILE DI LOCK

Una *variabile di lock* è una variabile a due stati:

- aperto → 0
- chiuso → 1

ISTRUZIONI INDIVISIBILI

Per cambiare il valore del lock, vengono utilizzate particolari istruzioni macchina, che prendono il nome di *istruzioni indivisibili*.

USO DELLA VARIABILE DI LOCK

come detto, una *variabile di lock* è una variabile a due stati – aperto e chiuso – utilizzata dai processi per accedere alle sezioni critiche.

Quando un processo vuole eseguire una sezione critica, legge il contenuto della variabile di lock e:

- se il lock è aperto (valore 0), il processo imposta il valore del lock a chiuso (valore 1) ed esegue la sezione critica, dopodiché, al termine, lo imposta dinuovo ad aperto
- se il lock è chiuso (1), il processo attende che diventi aperto (0)

L'uso di questa variabile presenta un problema: supponiamo che un processo legga la variabile di lock e vede che è aperta (0), ma prima che possa chiuderla (1), viene schedulato un altro processo che va in esecuzione e chiude la variabile lock (1); quando il primo processo ritorna in esecuzione, chiuderà anch'esso la variabile lock (1) e i due processi saranno contemporaneamente nella sezione critica.

Per evitare race condition nell'impostazione della variabile di lock, viene utilizzata un'operazione indivisibile per la lettura e la chiusura.

Di seguito, illustriamo l'uso di due istruzioni indivisibili, chiamate istruzioni *test-and-set* e *swap*.

ISTRUZIONE TEST-AND -SET

L'istruzione TSL è un'istruzione indivisibile che implementa il seguente algoritmo atomicamente:

```
function test-and-set (var i : integer ) : boolean
begin
  if i = 0 then
    begin
      i:=1
      test-and-set := true
    end
  else test-and-set = false
```

La CPU che esegue l'istruzione TSL blocca il bus di memoria per impedire che altre CPU accedano alla memoria. Per usare l'istruzione, useremo una variabile condivisa, *lock*, per coordinare l'accesso alla memoria condivisa. Quando *lock* è a 0, qualunque processo può metterla a 1 usando l'istruzione TSL e poi leggere o scrivere nella memoria condivisa; quando ha finito, il processo mette *lock* dinuovo a 0.

In pratica, l'istruzione TSL legge il valore dell'argomento che gli viene passato. Se vale 0 lo rimpiazza con 1 e ritorna vero altrimenti lascia il valore inalterato e ritorna falso.

7.6.2 Proprietà dell'approccio hardware

VANTAGGI

E' un tipo di approccio che si può applicare su sistemi con un qualsiasi numero di processori.

E' semplice e facile da verificare

Garantisce la mutua esclusione

SVANTAGGI

Gli svantaggi sono legati ai due fenomeni della starvation e del busy waiting.

C'è il fenomeno del *busy waiting* in quanto i processi che attendono di eseguire la sezione critica, restano in attesa attiva, quindi la CPU è utilizzata inutilmente per i cicli di attesa.

Inoltre, questo approccio potrebbe soffrire anche del fenomeno della *starvation* in quanto non è garantita l'attesa limitata: quando un processo esce dalla sezione critica, la scelta del prossimo processo è arbitraria, quindi un processo potrebbe non essere mai scelto.

7.7 Approccio software

Gli approcci software si basano su algoritmi che tendono a garantire le condizioni necessarie per una corretta gestione delle sezioni critiche.

Questo approccio utilizza l'attesa attiva per ritardare un processo e fanno uso di una complessa organizzazione di condizioni logiche per assicurare l'assenza di race condition.

Essi non utilizzano alcuna caratteristica dell'architettura del computer, dei linguaggi di programmazione o del kernel per ottenere la mutua esclusione. Per questo motivo gli approcci algoritmici sono indipendenti dalle piattaforme hardware e software.

Dal momento che gli approcci algoritmici lavorano indipendentemente dal kernel, non possono adottare l'approccio bloccante per la sincronizzazione dei processi, ecco perché questo approccio si basa sui cicli ed è affetto da tutti i problemi che ne derivano.

7.7.1 Algoritmi per due processi

PRIMO TENTATIVO

Algoritmo 6.1 Primo tentativo

```

var      turn : 1..2;
begin
    turn := 1;
Parbegin
    repeat
        while turn = 2
            do { niente };
            { Sezione critica }
            turn := 2;
            { Resto del ciclo }
    forever;
Parend;
end.

```

Processo P_1

```

repeat
    while turn = 1
        do { niente };
        { Sezione critica }
        turn := 1;
        { Resto del ciclo }
    forever;

```

Processo P_2

La variabile *turn* è una variabile condivisa che indica quale sarà il prossimo processo ad accedere alla sezione critica. Essa può assumere due valori, o il valore 1 o il valore 2. Prima che i due processi, P1 e P2, vengano creati, viene inizializzata a 1.

Il processo P1 può entrare nella sezione critica se *turn*=1, mentre P2 può entrarci se *turn*=2.

FUNZIONAMENTO:

Supponiamo che P1 voglia entrare in sezione critica. Controlla il valore della variabile *turn*.

Se *turn*=1 entra in sezione critica, esegue le operazioni e una volta che le ha terminate, esce dalla sezione e imposta *turn*=2 in modo tale da permettere a P2 di entrare in sezione critica.

Supponiamo che P2 voglia entrare in sezione critica. Controlla il valore della variabile *turn*.

Se $turn=2$ entra in sezione critica, esegue le operazioni e una volta che le ha terminate, esce dalla sezione e imposta $turn=1$ in modo tale da permettere a P1 di entrare in sezione critica.

In questo modo è garantita la mutua esclusione.

CONSIDERAZIONI:

Questo algoritmo soffre del fenomeno del *busy waiting*. In pratica, un processo non può fare niente di produttivo (continua a ciclare) finché non ha il permesso di entrare in sezione critica; il processo controlla periodicamente la variabile $turn$ e, perciò, consuma tempo di CPU, rimanendo attivo mentre aspetta.

Questa soluzione garantisce la mutua esclusione, ma porta tre conseguenze negative:

- possibilità di avere starvation in quanto non viene soddisfatta la proprietà dell'*attesa limitata*, poiché quando molti processi sono in attesa attiva per una sezione critica, non si può determinare quale di questi processi andrà in sezione critica quando il processo che sta in sezione critica termina
- se un processo fallisce, l'altro processo rimane bloccato per sempre; questo accade sia se il processo fallisce nella sua sezione critica sia se fallisca fuori. Questo viola una delle proprietà, in particolare quella chiamata *progresso*
- i processi si alternano in senso stretto anche se la loro velocità è molto diversa; la velocità di esecuzione è data dal processo più lento

SECONDO TENTATIVO

Algoritmo 6.2 Secondo tentativo

```

var   c1, c2 : 0..1;
begin
    c1 := 1;
    c2 := 1;
Parbegin
    repeat
        while c2 = 0
            do {niente};
        c1 := 0;
        { Sezione critica }
        c1 := 1;
        { Resto del ciclo }
    forever;
Parend;
end.

```

Processo P₁

```

repeat
    while c1 = 0
        do {niente};
    c2 := 0;
    { Sezione critica }
    c2 := 1;
    { Resto del ciclo }
forever;

```

Processo P₂

Il problema del primo tentativo è che si mantengono informazioni riguardo l'identificatore del processo che può accedere alla sezione critica, ma in realtà sarebbe utile mantenere le informazioni sullo stato di entrambi i processi. In effetti, ogni processo dovrebbe avere la propria chiave della sezione critica, così che, se un processo fallisce l'altro può ancora accedere alla propria sezione critica.

In questo algoritmo vengono utilizzati due variabili, $c1$ e $c2$, che possono essere considerate rispettivamente come flag di stato per i processi P1 e P2.

Il processo P1 imposta $c1=0$ quando entra nella sua sezione critica e lo reimposta a 1 quando esce. In questo modo, il valore 0 indica che P1 è in sezione critica mentre il valore 1 indica che non è in sezione critica. Analogamente, il valore di $c2$ indica se P2 si trova o meno in sezione critica.

FUNZIONAMENTO:

Prima di entrare in sezione critica, ogni processo controlla se l'altro processo è in sezione critica. In caso positivo continua a ciclare finché l'altro processo non esce dalla sezione critica. In caso negativo, entra in sezione critica.

CONSIDERAZIONI:

La violazione della condizione di progresso, in questo algoritmo, viene eliminata.

Se un processo fallisce all'esterno della sua sezione critica, compreso il codice per modificare il flag, allora l'altro processo non è bloccato: infatti, l'altro processo può entrare nella propria sezione critica tutte le volte che vuole, perché il flag dell'altro processo è sempre falso.

Se un processo fallisce all'interno della propria sezione critica, o dopo aver messo il flag a vero prima di entrare, allora l'altro processo è bloccato per sempre.

Questo algoritmo, inoltre, non garantisce neppure la mutua esclusione quando i due processi tentano di entrare allo stesso tempo nelle rispettive sezioni critiche. Sia $c1$ che $c2$ saranno pari a 1 (poiché nessuno dei due processi si trova in sezione critica) per cui entrambi i processi entreranno nelle sezioni critiche.

Per evitare questo problema, le istruzioni "while $c2 = 0$ do { niente };" e " $c1 := 0$;" nel processo P1 possono essere invertite. Analogamente, le istruzioni "while $c1 = 0$ do { niente };" e " $c2 := 0$;" nel processo P2 possono essere invertite. In questo modo $c1$ sarà impostato a 0 prima che P1 controlli il valore di $c2$ e dunque entrambi i processi non potranno trovarsi nelle rispettive sezioni critiche allo stesso tempo.

Tuttavia, se entrambi i processi tentano di entrare in sezione critica allo stesso tempo, sia $c1$ che $c2$ saranno a 0 per cui entrambi i processi aspetteranno l'un l'altro indefinitamente. Questa situazione prende il nome di **stallo (o deadlock)**.

**ALGORITMO DI DEKKER****Algoritmo 6.3 Algoritmo di Dekker**

```

var   turn : 1..2;
       c1, c2 : 0..1;
begin
    c1 := 1;
    c2 := 1;
    turn := 1;
Parbegin
    repeat
        c1 := 0;
        while c2 = 0 do
            if turn = 2 then
                begin
                    c1 := 1;
                    while turn = 2
                        do { niente };
                    c1 := 0;
                end;
                { Sezione critica }
                turn := 2;
                c1 := 1;
                { Resto del ciclo }
            forever;
Parend;
end.
Processo P1
repeat
    c2 := 0;
    while c1 = 0 do
        if turn = 1 then
            begin
                c2 := 1;
                while turn = 1
                    do { niente };
                c2 := 0;
            end;
            { Sezione critica }
            turn := 1;
            c2 := 1;
            { Resto del ciclo }
        forever;
Processo P2

```

Con la soluzione precedente abbiamo visto che bisogna osservare lo stato di entrambi i processi ma come si è dimostrato questo non è sufficiente. L'algoritmo di Dekker combina le soluzioni adottate nei due casi precedenti. Per evitare i vari problemi è necessario imporre un ordine di attività dei due processi.

Se c'è competizione per l'entrata in sezione critica, si può usare la variabile *turn* del primo tentativo per indicare a qual processo dovrebbe essere consentita l'entrata in sezione critica. Nel caso in cui non ci sia competizione, *turn* non ha alcun effetto.

Le variabili $c1$ e $c2$ sono usate come flag di stato proprio come nel secondo tentativo.

FUNZIONAMENTO:

La variabile *turn* è utile quando c'è competizione per l'entrata in sezione critica dei due processi. Essa indicherà quale dei due processi può entrare in sezione critica.

Se non c'è competizione per l'entrata, *turn* non ha alcun effetto.

Le variabili $c1$ e $c2$ sono usate come flag di stato dei due processi. In pratica indicano se il processo ha intenzione o meno di entrare in sezione critica.

L'istruzione $c1=0$ è messa prima del while per evitare il problema descritto nel secondo tentativo.

Quando il processo P1 vuole entrare in sezione critica, mette il proprio flag ($c1$) a vero, cioè pone $c1=0$.

L'istruzione successiva, "while $c2 = 0$ do" serve per controllare se anche l'altro processo vuole entrare in sezione critica.

Se è falso ($c2=1$) allora P1 può entrare in sezione critica saltando il ciclo while.

In caso contrario, se $c2=0$, il processo entra nel while. Ciò significa che c'è competizione per l'entrata in sezione critica ed entra in gioco il valore di *turn*.

In questo ciclo, il processo P1 effettua un nuovo controllo, verifica il valore della variabile *turn*. Il valore di questa variabile indica il processo che deve essere favorito per l'entrata in sezione critica: se è 1, allora deve essere favorito P1, se è 2 allora deve essere favorito il processo P2.

Quindi, se $turn=1$, P1 salta l'IF ed entra in sezione critica.

Se, invece, $turn=2$, P1 deve rinunciare ad entrare in sezione critica e deve favorire P2 per l'accesso in sezione critica, quindi imposta il suo flag $c1=1$ permettendo a P2 di entrare. Quindi si pone in attesa attiva fino a che il processo P2 non esce dalla sua sezione critica.

Dopo aver usato la propria sezione critica, P1 imposta $turn=2$ per trasferire a P2 il diritto di entrare in sezione critica ed imposta il suo flag $c1=1$ per liberare la propria sezione critica.

CONCLUSIONI:

Mediante questi accorgimenti, l'algoritmo soddisfa la mutua esclusione ed inoltre evita la situazione di *deadlock*. Il valore attuale di *turn* in ogni istante non è determinante per la correttezza dell'algoritmo.

ALGORITMO DI PETERSON



Algoritmo 6.4 Algoritmo di Peterson

```

var      flag : array [0..1] of boolean;
          turn : 0..1;
begin
    flag[0] := false;
    flag[1] := false;
Parbegin
  repeat
    flag[0] := true;
    turn := 1;
    while flag[1] and turn = 1
      do { niente };
      { Sezione critica }
      flag[0] := false;
      { Resto del ciclo }
  forever;
Parend;
end.
  
```

Processo P_0

```

repeat
  flag[1] := true;
  turn := 0;
  while flag[0] and turn = 0
    do { niente };
    { Sezione critica }
    flag[1] := false;
    { Resto del ciclo }
  forever;
  
```

Processo P_1

L'algoritmo di Peterson, così come l'algoritmo di Dekker, risolve il problema della mutua esclusione ma in un modo più semplice rispetto all'algoritmo di Dekker.

La variabile *turn* indica a chi spetta entrare nella sezione critica, mentre l'array *flag* indica se un processo è pronto ad entrare nella propria sezione critica (cioè indica l'intenzione di entrare).

Utilizza, invece che variabili di *flag*, un array di *flag* contenente un *flag* per ogni processo (qui, visto che ci sono 2 processi, conterrà due *flag*). Il valore di ciascun *flag* è di tipo booleano, *true* o *false*.

Un processo imposta il suo *flag* a *true* quando vuole entrare in sezione critica, mentre lo imposta a *false* quando esce dalla sezione critica.

Si assume che i due processi abbiano ID P0 e P1. L'ID del processo è usato come indice per accedere al *flag* di stato di un processo nell'array *flag*.

FUNZIONAMENTO:

Consideriamo il processo P0. Se vuole entrare in sezione critica imposta $flag[0]=true$.

Dopodiché cerca di favorire l'altro processo impostando $turn=1$.

A questo punto P0 controlla se l'altro processo entra in sezione critica, cioè se il suo *flag* è impostato a *true* e se $turn=1$. Se queste condizioni si verificano, allora P0 continua a ciclare (attesa attiva). Altrimenti entra in sezione critica.

Se entrambi i processi, P0 e P1, vogliono entrare in sezione critica, il valore di *turn* determina quale processo può entrare nella propria sezione critica.

All'uscita della sezione critica, il processo P0 imposta $flag[0]=false$ per permettere all'altro processo di poter accedere in sezione critica.

Un ragionamento analogo è valido nel caso in cui si consideri il funzionamento di P1.

CONCLUSIONI:

Questo algoritmo garantisce la mutua esclusione ed, inoltre, non è possibile che i due processi si blocchino a vicenda: se P0 è bloccato nel suo ciclo while, allora $flag[1]=true$ e $turn=1$; P0 può entrare nella propria sezione critica quando $flag[1]=false$, oppure quando $turn=0$.

Questo algoritmo fornisce una soluzione semplice al problema della mutua esclusione per due processi; inoltre può essere generalizzato al caso di n processi.

7.7.2 Algoritmi per n processi

Se si utilizza l'approccio algoritmico per implementare una sezione critica, l'algoritmo deve conoscere il numero di processi che utilizzano la sezione critica per gli stessi dati.

Questa conoscenza è utile per molte circostanze:

- la dimensione dell'array dei flag di stato
- i controlli per determinare se qualche altro processo desidera entrare in una sezione critica

Ognuna di queste caratteristiche deve essere modificata se cambia il numero di processi che devono essere gestiti dalla sezione critica.

Ad esempio, se si prendono in considerazione 2 processi, ogni processo deve controllare lo stato di un solo processo e nel caso favorirlo per l'esecuzione, per assicurare la correttezza e l'assenza di deadlock. Se si prendono in considerazione n processi, ogni processo deve controllare lo stato di $n-1$ altri processi e deve farlo in modo da prevenire race condition. Queste ragioni rendono l'algoritmo per n processi più complesso.

ALGORITMO DEL PANETTIERE / FORNAIO (BAKERY)

Algoritmo 6.6 Algoritmo del panettiere (Bakery) (Lamport [1974])

```

const n = ...;
var choosing : array [0..n - 1] of boolean;
      number : array [0..n - 1] of integer;
begin
  for j := 0 to n - 1 do
    choosing[j] := false;
    number[j] := 0;
Parbegin
  process  $P_i$  :
    repeat
      choosing[i] := true;
      number[i] := max (number[0],...,number[n - 1])+1;
      choosing[i] := false;
      for j := 0 to n - 1 do
        begin
          while choosing[j] do { nothing };
          while number[j] ≠ 0 and (number[j], j) < (number[i], i)
            do { nothing };
        end;
        { Sezione critica }
        number[i] := 0;
        { Resto del ciclo }
      forever;
      processo  $P_j$  : ...
Parend;
end.

```

Questo algoritmo è stato sviluppato nel 1974 ed ha lo scopo di risolvere il problema della sezione critica per n processi. È noto come **algoritmo del panettiere** ed è basato su uno schema di servizio comunemente usato nelle panetterie.



FUNZIONAMENTO:

Al suo ingresso nel negozio, ogni cliente riceve un numero; si serve, progressivamente, il cliente con numero più basso. Poiché l'algoritmo del panettiere non può assicurare che due processi (clienti) non ricevano lo stesso numero, a parità di numero si serve il processo con nome minore. In pratica, se P_i e P_j ricevono lo stesso numero e $i < j$, si serve prima P_i . Poiché i nomi dei processi sono unici e totalmente ordinati, l'algoritmo è del tutto deterministico.

Le strutture dati comuni sono:

- $choosing[i]$ è un array di flag booleane che indica se il processo P_i in quell'istante è impegnato nella scelta di un numero;
- $number[i]$ che contiene il numero del processo P_i ; $number[i]=0$ se P_i non ha scelto un numero dall'ultima volta in cui è entrato in sezione critica.

In pratica, nei due cicli while si controllano se ci sono processi che hanno un numero minore del processo corrente, ed eventualmente, a parità di numero, se ci sono processi che hanno un ID minore. In questo modo, un processo che vuole entrare in sezione critica favorisce un processo con un numero minore ed, eventualmente, con ID minore.

7.8 Approccio SO/Linguaggio – Semafori

Le soluzioni che si basano sull'approccio basato su linguaggio di programmazione usano meccanismi di IPC come i semafori, i monitor o lo scambio di messaggi.

I **meccanismi di IPC** (*InterProcess Communication*) sono essenzialmente:

- semafori – variabili su cui sono definite operazioni indivisibili
- monitor – costrutti linguistici che permettono di incapsulare le sezioni di codice che hanno funzioni di comunicazione e sincronizzazione
- scambio di messaggi – meccanismi di comunicazione stile I/O per il trasferimento di informazioni tra due o più processi

7.8.1 Semafori

Un **semaforo** è una particolare struttura di sincronizzazione. Consiste in una variabile intera condivisa a valori non negativi che può essere soggetta solo alle seguenti operazioni:

1. inizializzazione (specificata come parte della sua dichiarazione)
2. le operazioni indivisibili *wait* e *signal*

Il principio fondamentale è il seguente: due o più processi possono comunicare attraverso semplici segnali, in modo tale da sincronizzarsi tra loro. Le istruzioni usate per realizzare questa comunicazione sono la *wait* e la *signal*.

```

procedure wait (S)
begin
  if S > 0
    then S := S-1;
    else blocca il processo su S;
end;

procedure signal (S)
begin
  if qualche processo è bloccato su S
    then attiva un processo bloccato;
    else S := S+1;
end;
  
```

Le due operazioni utilizzate sono *wait* e *signal* e nella figura è illustrata la loro semantica.

L'indivisibilità delle operazioni *wait* e *signal* è assicurata dal linguaggio di programmazione o dal SO che le implementa. In pratica, le operazioni visibili in figura sono eseguite atomicamente.

Quando un processo effettua una *wait* su un semaforo, controlla se il valore del semaforo è > 0 . In caso affermativo, decrementa il valore del semaforo e consente al processo di proseguire la sua esecuzione; altrimenti, blocca il processo sul semaforo.

Un'operazione *signal* su un semaforo controlla se ci sono processi bloccati sul semaforo. In caso affermativo, l'operazione attiva un processo bloccato sul semaforo; altrimenti incrementa il valore del semaforo di un'unità.

7.8.2 Uso dei semafori nei sistemi concorrenti

Uso	Descrizione
Mutua esclusione	La mutua esclusione può essere implementata usando un semaforo inizializzato a 1. Un processo effettua un'operazione <i>wait</i> sul semaforo prima di entrare in una CS e un'operazione <i>signal</i> al termine della CS. Uno speciale tipo di semaforo chiamato <i>semaforo binario</i> semplifica ulteriormente l'implementazione della CS.
Concorrenza limitata	La concorrenza limitata implica che una funzione può essere eseguita, o una risorsa può essere utilizzata, da n processi concorrentemente, $1 \leq n \leq c$, con c costante. Un semaforo inizializzato a c può essere utilizzato per implementare la concorrenza limitata.
Segnalazione	La segnalazione viene usata quando un processo P_i vuole effettuare un'operazione a_i solo dopo che il processo P_j ha effettuato una operazione a_j . Viene implementata utilizzando un semaforo inizializzato a 0. P_i effettua una <i>wait</i> sul semaforo prima di effettuare l'operazione a_i . P_j effettua una <i>signal</i> sul semaforo dopo aver effettuato l'operazione a_j .

La tabella riassume tre utilizzi dei semafori nell'implementazione dei sistemi concorrenti.

La *mutua esclusione* è utile per implementare le sezioni critiche.

La *concorrenza limitata* è importante quando una risorsa può essere condivisa da al più c processi ($c \geq 1$).

La *segnalazione* è utile nel controllo della sincronizzazione.

MUTUA ESCLUSIONE

```
var sem_CS : semaforo := 1;
Parbegin
repeat
    wait (sem_CS);
    { Sezione critica }
    signal (sem_CS);
    { Parte restante del ciclo }
all'infinito;
Parend;
end.

Processo Pi                                Processo Pj
```

La figura mostra l'implementazione di una sezione critica nei processi P_i e P_j usando un semaforo chiamato *sem_CS* inizializzato a 1.

Ogni processo effettua:

- una *wait* sul semaforo prima di entrare nella propria sezione critica
- una *signal* al termine della sezione critica

FUNZIONAMENTO:

Il primo processo a effettuare la *wait* trova che *sem_CS* > 0 quindi decrementa il valore del semaforo di un'unità ed entra nella sua sezione critica.

Quando il secondo processo effettua la *wait*, si blocca sul semaforo perché il suo valore è 0. Viene riattivato quando il primo processo effettua la *signal* una volta che esce dalla sua sezione critica. A questo punto il secondo processo può entrare nella sua sezione critica.

Se non ci sono processi bloccati quando il primo processo effettua la *signal*, allora il valore del semaforo *sem_CS* diventa 1, questo valore permette ad un processo, che successivamente effettua una *wait*, di entrare immediatamente nella sua sezione critica.

CONCLUSIONI:

La condizione di *mutua esclusione* è garantita visto che il semaforo è inizializzato a 1.

La condizione di *progresso* è garantita in quanto un processo che effettua una *wait* ottiene l'accesso alla sua sezione critica se non c'è un altro processo nella propria sezione critica.

La condizione di *attesa limitata* non è verificata in quanto potrebbe verificarsi il fenomeno della *starvation*.

Inoltre, essendo *wait* e *signal* due funzioni primitive, un loro uso scorretto porterebbe a problemi di correttezza:

- se si sostituisse alla *wait* una *signal*, cioè "*signal* -> *CS* -> *signal*", più processi contemporaneamente potrebbero accedere alle proprie sezioni critiche
- se si sostituisse alla *signal* una *wait*, cioè "*wait* -> *CS* -> *wait*", un processo, all'uscita dalla sua sezione critica, rimarrebbe bloccato così come gli altri processi che vorrebbero accedere in sezione critica, generando quindi una situazione di *deadlock*

SEMAFORI BINARI (MUTEX)

In alcune occasioni può essere utilizzata una versione semplificata dei semafori, detta *mutex* (*semaforo binario*). Un semaforo binario è un particolare semaforo usato per implementare la mutua esclusione.

Esso è inizializzato a 1 ma può assumere solo i valori 0 e 1 durante l'esecuzione del programma.

Le operazioni *wait* e *signal* sono differenti rispetto a quelle eseguite su un semaforo normale:

- l'istruzione "*S = S - 1*" è sostituita dall'istruzione "*S = 0*"
- l'istruzione "*S = S + 1*" è sostituita dall'istruzione "*S = 1*"

Ogni volta che un processo ha bisogno di accedere ai dati condivisi, acquisisce il mutex (*mutex_lock*); quando l'operazione è terminata, il mutex viene rilasciato (*mutex_unlock*), permettendo ad un altro processo di acquisirlo per eseguire le sue operazioni.

CONCORRENZA LIMITATA

La *concorrenza limitata* implica che un'operazione possa essere eseguita da al più c processi (con $c \geq 1$). La concorrenza limitata viene implementata inizializzando un semaforo *sem_C* a c .

Il valore del semaforo rappresenta:

- il numero di accessi consentiti (risorse libere) se ≥ 0
- il numero di processi in attesa se < 0 (in valore assoluto)

I semafori utilizzati per implementare la concorrenza limitata vengono denominati *semafori contatore*.

SEGNALAZIONE TRA PROCESSI

La *segnalazione* viene usata quando un processo *Pi* vuole effettuare un'operazione (*a_i*) solo dopo che un processo *Pj* ha effettuato un'altra operazione (*a_j*).

Viene implementata utilizzando un semaforo inizializzato a 0. *Pi* effettua una *wait* sul semaforo prima di effettuare l'operazione *a_i*. *Pj* effettua una *signal* sul semaforo dopo aver effettuato l'operazione *a_j*.

7.9 Approccio SO/Linguaggio – Monitor

Un **monitor** è un costrutto di sincronizzazione che può essere utilizzato da due o più processi per rendere mutuamente esclusivo l'accesso a risorse condivise. Esso è un modulo software che contiene una o più funzioni e procedure, una sequenza di inizializzazione e variabili locali.

Inoltre può contenere le dichiarazioni di speciali dati di sincronizzazione chiamati **variabili di condizione** su cui possono essere effettuate solo le operazioni *wait* e *signal*.

I processi possono chiamare le procedure di un monitor ogni volta che vogliono ma non possono accedere direttamente alle variabili locali del monitor.

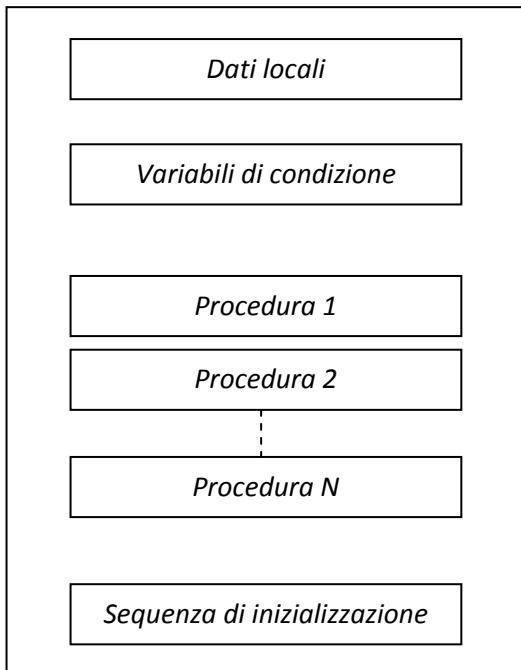
I monitor possiedono un'importante *proprietà* che li rende utili per ottenere la mutua esclusione: ad ogni istante, un solo processo può essere attivo in un monitor. Quando un processo chiama una procedura di monitor, le prime istruzioni della procedura controlleranno se un qualche altro processo è correntemente attivo dentro il monitor; se è così, il processo chiamante sarà sospeso finché l'altro processo non avrà lasciato il monitor. Se nessun altro processo sta usando il monitor, il processo chiamante può entrare.

Riassumendo, le caratteristiche principali di un monitor sono le seguenti:

1. le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne
2. un processo entra nel monitor chiamando una delle sue procedure
3. solo un processo alla volta può essere in esecuzione all'interno del monitor; ogni altro processo che ha chiamato il monitor è sospeso, nell'attesa che questo diventi disponibile

Le chiamate delle operazioni sono servite in ordine FIFO per soddisfare la proprietà di attesa limitata. In pratica, la coda di processi entranti è di tipo FIFO.

7.9.1 Struttura di un monitor



```
monitor assegnazione_risorse
{
    boolean occupato;
    condition x;

    void acquisizione(int tempo) {
        if (occupato)
            x.wait(tempo);
        occupato = true;
    }

    void rilascio() {
        occupato = false;
        x.signal();
    }

    void inizializzazione() {
        occupato = false;
    }
}
```

La figura in alto a sinistra illustra la struttura di un monitor.

La figura in alto a destra illustra un esempio di corpo di un monitor.

7.9.2 Variabili di condizione

Sebbene i monitor forniscano una maniera semplice per ottenere la mutua esclusione, grazie alla loro proprietà, questo non è sufficiente, perché abbiamo bisogno di un modo per bloccare i processi quando non

possono proseguire. La soluzione sta nell'introduzione di *variabili di condizione*, accessibili solo dall'interno del monitor; quando una procedura di monitor scopre che non può continuare, chiama una *wait* su una variabile di condizione.

Una *variabile di condizione* è una variabile con l'attributo **condizione** ed è associata a una condizione nel monitor. Le uniche due operazioni eseguibili su una variabile di condizione sono *wait* e *signal*:

- la *wait* sospende l'esecuzione del processo chiamante sulla condizione *c*; il monitor diventa disponibile per gli altri processi
- la *signal* riattiva un processo sospeso sulla condizione *c*; se i processi sospesi sono molti, ne sceglie uno; se non ce ne sono, non fa niente

Queste due operazioni sono diverse da quelle dei semafori: se un processo in un monitor effettua una *signal* e non c'è nessun processo bloccato su quella variabile di condizione, il segnale viene perso. Inoltre, la *wait* e la *signal* del monitor non sono contatori.

L'implementazione di un monitor mantiene diverse code di processi, una per ogni variabile di condizione ed una per i processi in attesa di eseguire le operazioni del monitor. Per assicurare che i processi non restino bloccati durante l'esecuzione di un'operazione, il monitor favorisce i processi che sono stati attivati dalle operazioni *signal* rispetto a quelli che vogliono iniziare l'esecuzione delle operazioni del monitor.

7.9.3 Semafori vs. monitor

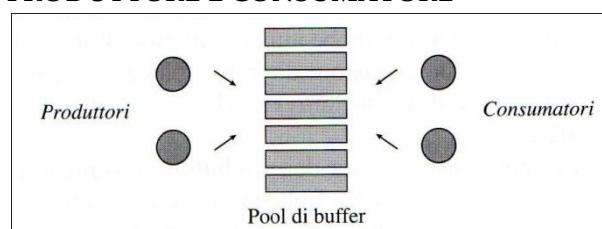
- Ad ogni semaforo è associata una coda di processi in attesa sul semaforo.
- Ad ogni variabile di tipo condition è associata una coda su cui attendono i processi sospesi
- L'operazione *wait* su un semaforo è sospensiva solo se il semaforo non è positivo
- L'operazione *wait* su un monitor è immediatamente sospensiva

7.10 Problemi classici di sincronizzazione dei processi

Come discusso nei paragrafi precedenti, le sezioni critiche e l'utilizzo dei segnali rappresentano gli elementi chiave per la sincronizzazione dei processi, per cui una soluzione a un problema di sincronizzazione dovrebbe incorporare una combinazione adeguata di questi elementi.

In questo paragrafo, analizzeremo alcuni problemi classici di sincronizzazione dei processi. Esiste, infatti, una serie di problemi classici nella programmazione concorrente che vengono utilizzati per dimostrata l'efficienza di determinate teorie od algoritmi e forniscono una base comune per poter effettuare dei paragoni.

PRODUTTORE E CONSUMATORE



Questo problema è anche noto come *problema del buffer a capienza limitata*.

Esso si compone di un numero non specificato di *produttori* e *consumatori*, e di un insieme finito di *buffer*, ciascuno dei quali è in grado di contenere un elemento di informazione.

Un *buffer* è *pieno* quando un produttore scrive un nuovo elemento al suo interno.

Un *buffer* è *vuoto* quando un consumatore estrae un elemento contenuto in esso.

Un *processo produttore* produce un elemento di informazione alla volta e lo inserisce in un buffer vuoto.

Un *processo consumatore* estrae l'informazione, un elemento alla volta, da un buffer pieno.

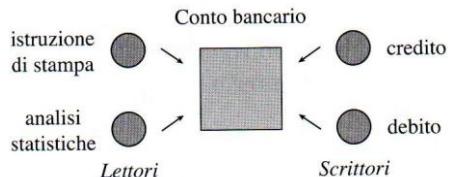
Una soluzione al problema produttori-consumatori deve soddisfare queste condizioni:

1. un produttore non deve sovrascrivere un buffer pieno
2. un consumatore non deve consumare un buffer vuoto
3. i produttori e i consumatori devono accedere ai buffer in maniera mutuamente esclusiva

ed alcune volte viene imposta anche la seguente condizione:

4. l'informazione deve essere consumata nello stesso ordine con la quale è stata inserita nel buffer, ovvero secondo l'ordine FIFO

LETTORI E SCRITTORI



Esso si compone di un numero non specificato di *lettori* e *scrittori*. Entrambi agiscono su dati differenti.

Un *processo lettore* può esclusivamente leggere i dati.

Un *processo scrittore* può modificare o aggiornare i dati.

Si utilizzeranno i termini *leggere* e *scrivere* per gli accessi ai dati

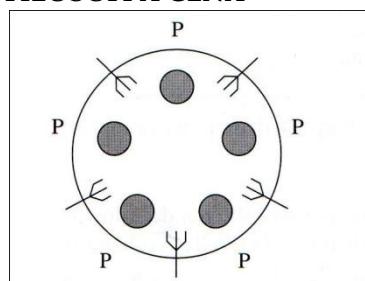
condivisi eseguiti, rispettivamente, dai processi lettori e scrittori.

Una soluzione al problema lettori-scrittori deve soddisfare queste condizioni:

1. molti lettori possono effettuare la lettura in maniera concorrente
2. solo uno scrittore per volta può eseguire la scrittura
3. la lettura è proibita quando uno scrittore sta eseguendo una scrittura

I punti 1-3 consentono o a uno scrittore di eseguire la scrittura o a più lettori di eseguire letture concorrenti.

FILOSOFI A CENA



Ci sono cinque filosofi seduti attorno a un tavolo. Ogni filosofo ha davanti a sé un piatto, mentre tra ogni coppia di filosofi c'è una forchetta. I filosofi alternano momenti durante i quali meditare e momenti durante i quali mangiare.

Per mangiare, un filosofo deve prendere, una alla volta, le due forchette che si trovano alla sua destra e alla sua sinistra. Durante la meditazione, invece, un filosofo deve tenere le forchette sul tavolo.

Risulta evidente che il numero di forchette impedisce a tutti i filosofi di mangiare contemporaneamente quindi una corretta programmazione concorrente deve essere in grado di far mangiare alternativamente tutti i filosofi evitando che qualcuno in particolare soffra di starvation ed evitando che si verifichino stalli in fase di "acquisizione delle forchette". In pratica:

- la condizione di *correttezza* è mantenuta se un filosofo affamato non attende indefinitamente quando decide di mangiare
- i *deadlock* sono evitati quando i processi non si bloccano rimanendo in attesa l'uno dell'altro
- i *livelock* sono evitati quando i processi non ritardano l'esecuzione a favore degli altri in modo indefinito

BARBIERE CHE DORME

Un barbiere possiede un negozio con una sola sedia da lavoro e un certo numero limitato di posti per attendere. Se non ci sono clienti il barbiere dorme; altrimenti, all'arrivo del primo cliente il barbiere si sveglia ed inizia a servirlo. Se dovessero sopraggiungere clienti durante il periodo di attività del barbiere, essi si mettono in attesa sui posti disponibili. Al termine dei posti di attesa, un ulteriore cliente viene scartato.

Questa problematica è molto vicina al sistema di funzionamento degli helpdesk informalizzati dove l'operatore serve, uno per volta, tutti i clienti in coda oppure attende, senza effettuare alcuna operazione in particolare, l'arrivo di nuove chiamate. Una corretta programmazione concorrente deve far "dormire" il barbiere in assenza di clienti, attivare il barbiere sul primo cliente al suo arrivo e mettere in coda tutti i successivi clienti tenendoli inattivi.

7.11 Alcune soluzioni

Di seguito verranno proposte alcune soluzioni ai problemi tipici riguardanti la sincronizzazione tra processi.

7.11.1 Prodottoconsumatore mediante sezioni critiche

```

begin
Parbegin
    var prodotto : boolean;
repeat
    prodotto := false
    while prodotto = false
        if esiste un buffer vuoto
        then
            { Inserisci nel buffer }
            prodotto := true;
            { Parte restante del ciclo }
        forever;
Parend;
end.

Produttore
  
```



```

var consumato : boolean;
repeat
    consumato := false;
    while consumato = false
        if esiste un buffer pieno
        then
            { Estrai dal buffer }
            consumato := true;
            { Parte restante del ciclo }
        forever;
  
```

Consumatore

La figura mostra una soluzione al problema dei produttori-consumatori utilizzando le sezioni critiche.

FUNZIONAMENTO:

Il produttore utilizza la variabile booleana PRODOTTO per interrompere il ciclo while dopo aver prodotto un elemento.

Il consumatore usa la variabile booleana CONSUMATO per interrompere il ciclo while che ha consumato un elemento.

Il produttore controlla ripetutamente se esistono buffer vuoti. Non appena ne trova uno, inserisce l'elemento nel buffer e imposta a *true* la variabile PRODOTTO.

Il consumatore controlla ripetutamente se esistono buffer pieni. Non appena ne trova uno estrae l'elemento dal buffer e imposta a *true* la variabile CONSUMATO.

PROBLEMI:

Questa soluzione ha due problemi:

- visto che entrambi gli accessi al buffer si trovano in sezioni critiche, allora un solo processo per volta, produttore o consumatore, può accedere al buffer in ogni istante di tempo anche se abbiamo a disposizione più buffer
- entrambi i processi vanno in attesa attiva quando controllano se ci sono buffer pieni e buffer vuoti



7.11.2 Produttore-consumatore mediante semafori

Il secondo problema, cioè quello legato all'attesa attiva può essere risolto utilizzando i semafori.

```

const           n = . . . ;
type            item = . . . ;
var
    buffer : array [0..n - 1] of item;
    full : Semaforo := 0; { Inizializzazioni }
    empty : Semaforo := n;
    prod_ptr, cons_ptr : integer;
begin
    prod_ptr := 0;
    cons_ptr := 0;
Parbegin
    repeat           repeat
        wait (vuoto);
        buffer [prod_ptr] := . . . ;
        { i.e. produci }
        prod_ptr := prod_ptr + 1 mod n;
        signal ( pieno );
        { Parte restante del ciclo }
    all'infinito;
Parend;
end.
Produttore          Consumatore

```

La figura mostra come i semafori possono essere usati per implementare una soluzione del problema produttore-consumatore con n buffer, un processo produttore e un processo consumatore.

Il pool di buffer è rappresentato da un array di buffer con un singolo elemento all'interno.

Vengono dichiarati due semafori, *pieno* e *vuoto*. I valori dei semafori *vuoto* e *pieno* indicano il numero di buffer, rispettivamente, vuoti e pieni, per cui sono inizializzati, rispettivamente, a n e a 0.

prod_ptr e *cons_ptr* sono usati come indici dell'array *buffer* e sono inizializzati a 0.

Funzionamento:

Il *produttore* aspetta un buffer vuoto con una *wait(vuoto)*. Quando sono disponibili buffer vuoti, inserisce un elemento nel buffer e aggiorna l'indice *prod_ptr*. Dopo aver completato l'operazione di inserimento effettua una *signal(pieno)* per permettere al consumatore di entrare nella sua sezione critica.

Il *consumatore* aspetta un buffer pieno con una *wait(pieno)*. Quando sono disponibili buffer pieni, estrae un elemento dal buffer e aggiorna l'indice *cons_ptr*. Dopo aver completato l'operazione di estrazione effettua una *signal(vuoto)* per permettere al produttore di entrare nella sua sezione critica.

Considerazioni:

L'uso dei segnali *wait* e *signal* permette di evitare le attese attive poiché i semafori sono utilizzati per controllare i buffer pieni e vuoti, per cui un processo sarà bloccato se non trova un buffer vuoto o pieno come richiesto.

Il livello di concorrenza in questo algoritmo è 1, a volte viene eseguito un produttore, altre volte viene eseguito un consumatore.

L'algoritmo scritto in questo modo assicura che i buffer siano utilizzati in ordine FIFO.

7.11.3 Produttore-consumatore mediante monitor

```

type Bounded_buffer_type = monitor
  const
    n = . . .;                                { Numeri di buffer }
  type
    item = . . .;
  var
    buffer : array [0..n-1] of item;
    full, prod_ptr, cons_ptr : integer;
    buff_full : condition;
    buff_empty : condition;
  procedura produci (produced_info : item);
  begin
    if full = n then buff_empty.wait;
    buffer [prod_ptr] := produced_info;           { i.e. produci }
    prod_ptr := prod_ptr + 1 mod n;
    full := full + 1;
    buff_full.signal;
  end;
  procedura consuma (for_consumption : item);
  begin
    if full = 0 then buff_full.wait;
    for_consumption := buffer[cons_ptr];          { i.e. consuma }
    cons_ptr := cons_ptr + 1 mod n;
    full := full - 1;
    buff_empty.signal;
  end;
  begin { inizializzazione }
    full := 0;
    prod_ptr := 0;
    cons_ptr := 0;
  end;
  begin
    var B_buf : Bounded_buffer_type;
  Parbegin
    var info : item;                         var info : item;                         var area : item;
    repeat                                repeat                                repeat
      info := . . .;                      info := . . .;                      B_buf.consume (area);
      B_buf.produce (info);               B_buf.produce (info);               { Area di consumo }
      { Parte restante                  { Parte restante
        del ciclo }                   del ciclo }                   { Parte restante
        all'infinito;                  all'infinito;                  del ciclo }
    Parend;
  end.

```

Produttore P₁Produttore P₂Consumatore P₃

La figura mostra una soluzione al problema produttori-consumatori mediante l'utilizzo dei monitor. Segue grossomodo lo stesso approccio della soluzione che utilizza i semafori.

Nella parte superiore c'è un tipo monitor *Bounded_buffer_type*.

Il pool di buffer è rappresentato da un array di buffer.

prod_ptr e *cons_ptr* sono usati come indici dell'array *buffer* e sono inizializzati a 0.

La variabile *pieno* indica il numero di buffer pieni.

buff_full e *buff_empty* sono variabili di condizione.

FUNZIONAMENTO:

Nella procedura del *produttore*, *produce*, un produttore controlla se tutti i buffer sono pieni, in questo caso (se *pieno=n*) esegue una *buff_empty.wait*. Altrimenti entra in sezione critica, inserisce un elemento in un buffer vuoto ed incrementa il numero di buffer pieni.

Analogamente, nella procedura del *consumatore*, *consuma*, un consumatore esegue una *buff_full.wait* se *pieno=0*, cioè se tutti i buffer sono vuoti. Altrimenti entra in sezione critica, estraie un elemento da un buffer pieno ed incrementa il numero di buffer vuoti.

I consumatori ed i produttori in attesa sono attivati, rispettivamente, dalle istruzioni *buff_full.signal* e *buff_empty.signal* che si trovano all'interno delle procedure *produce* e *consuma*.

Nella sequenza di inizializzazione del monitor, la variabile *pieno* e gli indici dell'array *prod_ptr* e *cons_ptr* sono inizializzati al valore 0.

CONSIDERAZIONI:

Nella parte bassa, c'è il codice inherente al processo produttore che ovviamente invoca la procedura *produce*, e il codice inherente al processo consumatore che ovviamente invoca la procedura *consuma*.

7.11.4 Lettore-scrittore con priorità ai lettori



```

program lettori_scrittori;
var numlettori: integer;
      mutex, dati: semaforo ( :=1 );

procedure lettore;
begin
  repeat
    wait (mutex);
    numlettori++;
    if numlettori = 1
      then wait (dati);
    signal (mutex);
    LEGGI;
    wait (mutex);
    numlettori--;
    if numlettori = 0
      then signal (dati);
    signal (mutex);
  forever
end;

```

```

procedure scrittore;
begin
  repeat
    wait (dati);
    SCRIVI;
    signal (dati);
  forever
end;

begin
  numlettori := 0;
  parbegin
    lettore;
    scrittore
  parend
end.

```

Per evitare race condition tutte le modifiche ai contatori vengono effettuate all'interno di sezioni critiche implementate usando un semaforo binario chiamato &mutex.

FUNZIONAMENTO:

Questa soluzione utilizza:

- la variabile condivisa *NUM_LETTORI* per memorizzare il numero di lettori ad ogni istante di tempo. Inoltre, grazie a questa variabile è possibile determinare qual è il primo lettore ad accedere ai dati e qual è l'ultimo a rilasciare i dati condivisi
- un semaforo *MUTEX* che serve per evitare inconsistenze quando si modifica il valore di *NUM_LETTORI*
- un semaforo *DATI* che serve per evitare che lettori e scrittori accedano contemporaneamente ai dati

Sia *MUTEX* che *DATI* sono semaforo binari inizializzati a 1.

Il lettore blocca il *MUTEX* per modificare e incrementare *NUM_LETTORI*. Se è il primo lettore, ferma gli scrittori mediante una *wait* sul semaforo *DATI*.

Dopo aver sbloccato il *MUTEX* accede in lettura ai dati.

Quindi riblocca il *MUTEX* per modificare *NUM_LETTORI*. Se è l'ultimo lettore, sblocca uno degli scrittori mediante una *signal* sul semaforo *DATI*. Quindi rilascia il *MUTEX*.

Lo scrittore non fa altro che bloccare i lettori quando esso sta scrivendo. Quando ha terminato risveglia un lettore o un altro scrittore.

CONSIDERAZIONI:

In questa soluzione i lettori hanno la priorità: quando un lettore inizia ad accedere ai dati, i lettori possono mantenere il controllo dell'area dati finché c'è un lettore attivo, quindi gli scrittori rischiano un'attesa perenne. In pratica, fino a che ci sono processi che vogliono leggere i dati, gli scrittori devono rimanere in attesa.

7.11.5 Lettore-scrittore con priorità agli scrittori



In questo paragrafo presentiamo la soluzione che impedisce a nuovi lettori di accedere ai dati se qualche scrittore ha dichiarato di voler effettuare una scrittura.

```
program lettori_scrittori;
var numlettori, numscrittori: integer;
    &mutex1, &mutex2: semaforo (:=1);
    &unoAllaVolta: semaforo (:=1);
    &scrittura, &lettura: semaforo (:=1);

procedure lettore;
begin
    repeat
        wait (unoAllaVolta);
        wait (lettura);
        wait (mutex2);
        numlettori++;
        if numlettori = 1
            then wait (scrittura);
        signal (mutex2);
        signal (lettura);
        signal (unoAllaVolta);
        LEGGI;
        wait (mutex2);
        numlettori--;
        if numlettori = 0
            then signal (&scrittura);
        signal (mutex2);
    forever
end;
```

```
procedure scrittore;
begin
    repeat
        wait (mutex1);
        numscrittori++;
        if numscrittori = 1
            then wait (lettura);
        signal (mutex1);
        wait (scrittura);
        SCRIVI;
        signal (scrittura);
        wait (mutex1);
        numscrittori--;
        if numscrittori = 0
            then signal (lettura);
        signal (mutex1);
    forever
end;

begin
    numlettori, numscrittori := 0;
    parbegin
        lettore;
        scrittore
    parend
end.
```

In questa soluzione ogni lettore che vuole leggere è accettato, a meno che uno scrittore chiede di entrare: in questo caso viene inibito l'ingresso ai lettori successivi.

FUNZIONAMENTO:

Questa soluzione usa:

- due variabili *NUM_LETTORI* e *NUM_SCRITTORI* per tenere traccia del numero di lettori e del numero di scrittori
- i semafori binari *MUTEX1* e *MUTEX2* per evitare inconsistenze quando si modificano rispettivamente le variabili *NUM_SCRITTORI* e *NUM_LETTORI*
- il semaforo binario *UNO_ALLA_VOLTA* per far sì che un solo lettore alla volta possa accodarsi su lettura
- il semaforo binario *LETTURA* che viene utilizzato dagli scrittori per bloccare i lettori
- il semaforo binario *SCRITTURA* che viene utilizzato dai lettori per impedire che lettori e scrittori accedano contemporaneamente ai dati

In pratica, ogni lettore che vuole leggere è accettato a meno che uno scrittore chiede di entrare: in questo caso viene inibito l'ingresso ai lettori successivi.

Il primo lettore inibisce gli scrittori ma consente ad altri lettori di leggere.

Il primo scrittore inibisce sia i lettori che gli scrittori.

SPIEGAZIONE LETTORE:

Quando un processo lettore vuole leggere effettua una wait su *UNO_ALLA_VOLTA*, dopodichè effettua la stessa operazione su *LETTURA*.

A questo punto non deve far altro che incrementare il valore di *NUM_LETTORI*, proteggendo la modifica di questa variabile bloccando il mutex.

Durante questa operazione, il processo controlla anche se è il primo lettore:

- se lo è, blocca eventuali processi scrittori che vogliono scrivere, ma permette ad altri processi lettori di leggere in modo concorrente gli stessi dati
- se non lo è, non fa nulla e continua eseguendo le operazioni successive

Una volta letti i dati, il processo deve decrementare il valore di *NUM_LETTORI*, proteggendo la modifica riboccando il mutex.

SPIEGAZIONE SCRITTORE:

Quando un processo scrittore vuole scrivere non deve far altro che incrementare il valore di *NUM_SCRITTORI* (bloccando il mutex) e controllare se è il primo scrittore:

- se lo è, allora deve bloccare tutti gli eventuali processi lettori che vogliono leggere i dati
- se non lo è, continua eseguendo le operazioni successive

Prima di poter scrivere, lo scrittore deve effettuare una wait su *SCRITTURA* in modo tale da controllare se c'è un altro processo scrittore che sta scrivendo.

Una volta effettuata la scrittura, occorre decrementare il valore di *NUM_SCRITTORI* (bloccando il mutex) e controllare se il processo è l'ultimo scrittore. Se lo è, sblocca uno dei lettori in attesa.

UTILITA' DEL SEMAFORO UNO ALLA VOLTA:

Questo semaforo torna utile quando ci sono uno o più lettori che stanno leggendo e viene accettato un processo (o più di uno) che vuole effettuare una scrittura.

Supponiamo che vi siano 3 lettori che stanno leggendo, 2 scrittori che vogliono scrivere e 2 lettori che vogliono leggere ma che arrivano dopo gli scrittori.

Quando il primo dei due scrittori entra, blocca i due processi lettori che vogliono leggere mediante l'operazione *wait (lettura)*, portando il valore del semaforo a 0.

In questo modo, il primo lettore che vuole leggere, effettua la wait su *UNO_ALLA_VOLTA* e assegna valore 0 a questo semaforo, dopodichè si blocca sull'operazione *wait (lettura)* perché il suo valore è già 0. Per questo motivo, tutti i successivi processi lettori si bloccheranno sul semaforo *UNO_ALLA_VOLTA* perché il valore di quest'ultimo è 0.

In questo modo, non appena l'ultimo dei 3 processi lettori termina di leggere, sblocca il primo scrittore che scrive, dopodichè anche il secondo scrittore scrive, e soltanto ora i due lettori verranno sbloccati e potranno leggere i dati.

7.11.6 Filosofi a cena

La condizione di correttezza nel sistema dei filosofi a cena è che un filosofo affamato non dovrebbe attendere indefinitamente quando decide di mangiare.

```

var      successful : boolean;
repeat
    successful := false;
    while (not successful)
        if entrambe le forchette sono disponibili then
            prendi le forchette una alla volta;
            successful := true;
        if successful = false
        then
            block ( $P_i$ );
            { mangia }
            posa entrambe le forchette;
            if il vicino di sinistra è in attesa della sua forchetta di destra
            then
                activate (vicino di sinistra);
            if il vicino di destra è in attesa della sua forchetta di sinistra
            then
                activate (vicino di destra);
            { pensa }
        all'infinito

```

Un filosofo controlla la disponibilità delle forchette in una sezione critica nella quale prendere anche le forchette. Per questo motivo non si possono verificare race condition.

Questa struttura assicura che almeno alcuni filosofi possono mangiare a ogni istante di tempo e previene il verificarsi di deadlock.

Un filosofo che non riesce a prendere entrambe le forchette allo stesso tempo si blocca. Verrà riattivato quando uno dei suoi vicini posa una forchetta condivisa; pertanto il processo bloccato deve controllare nuovamente la disponibilità delle forchette. Questo è lo scopo del ciclo *while*. Tuttavia, il ciclo causa una condizione di attesa attiva.

7.11.7 Filosofi a cena con monitor

```

monitor fc {
    enum {pensa, affamato, mangia} stato [5] ;
    condition auto [5];

    prende (int i) {
        stato[i] = affamato;
        verifica(i);
        if (stato[i] != mangia)
            auto[i].wait;
    }
    void posa (int i) {
        stato[i] = pensa;
        // verifica i commensali di destra e sinistra
        verifica((i + 4) % 5);
        verifica((i + 1) % 5);
    }
    void verifica (int i) {
        if( (stato[(i + 4) % 5] != mangia) &&
            (stato[i] == affamato) &&
            (stato[(i + 1) % 5] != mangia) ) {
            stato[i] = mangia ;
            auto[i].signal () ;
        }
    }
    void codice di inizializzazione() {
        for (int i = 0; i < 5; i++)
            stato[i] = pensa;
    }
}

```



Questa situazione è priva di situazioni di stallo ma un filosofo può attendere indefinitamente.

Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura di dati:

```
enum { pensa, affamato, mangia } stato [5];
```

Occorre ricordare che, per mangiare, un filosofo deve prendere entrambe le forchette ai suoi lati. Per questo motivo, un filosofo può mangiare solo se i suoi vicini non stanno mangiando.

FUNZIONAMENTO:

Il filosofo i può impostare la variabile $stato[i]=mangia$ solo se i suoi vicini non stanno mangiando.

```
(stato [ (i+4) % 5 ] != mangia) && (stato [ (i+1) % 5 ] != mangia)
```

Inoltre occorre impiegare la seguente struttura dati

```
condition auto [5];
```

dove il filosofo i può ritardare se stesso quando ha fame ma non riesce ad ottenere le forchette di cui ha bisogno. A questo punto si può descrivere la soluzione: la distribuzione delle forchette è controllata dal monitor fc . Ogni filosofo prima di cominciare deve invocare l'operazione *prendere*; ciò può determinare la sospensione del processo filosofo. Completata con successo questa operazione, il filosofo può mangiare; in seguito, il filosofo invoca *pensa*. Il filosofo i deve invocare in sequenza le operazioni *prendere* e *posa*:

```
fc.prendere(i); ... mangia ... fc.posa(i);
```

SPIEGAZIONE:

Il filosofo i -esimo richiama la procedura *PRENDE*. Qui cambia il suo stato in *affamato* per poi richiamare la procedura *VERIFICA*.

Nella procedura *VERIFICA*, il filosofo controlla se i suoi vicini stanno mangiando e se egli stesso vuole mangiare, cioè se il suo stato è *affamato*.

Se vuole mangiare e se i suoi vicini non stanno mangiando, allora prende le forchette e mangia. Viceversa, il filosofo non mangia, cioè il suo stato rimane inalterato.

Quindi, la procedura verifica, serve per modificare lo stato del filosofo i -esimo in *mangia* nel caso in cui si verifichino le condizioni elencate precedentemente.

A questo punto, il filosofo ritorna nella procedura *PRENDE* e controlla se il suo stato è cambiato.

Se è *mangia* allora passa direttamente alla procedura *POSA* per posare le forchette.

Se non è *mangia* allora si blocca sulla variabile di condizione *auto* in quanto non è riuscito a prendere le forchette per mangiare.

Dopo aver eseguito la procedura *PRENDE*, un filosofo esegue la procedura *POSA*. In questa procedura, il filosofo cambia il suo stato in *pensa*, dopodichè verifica prima se il suo vicino di sinistra può e vuole mangiare, poi verifica le stesse cose per il suo vicino di destra.

Un filosofo bloccato sulla variabile di condizione, viene risvegliato da uno dei suoi vicini, quando uno di questi richiama la procedura *VERIFICA* passando alla procedura stessa, il valore i del filosofo bloccato.

7.11.8 Barbiere addormentato con semafori



```

program barbiere_addormentato;
const sedie N;
var coda: integer;
      mutex: semaforo (:=1);
      cliente: semaforo (:=0);
      barbiere: semaforo (:=0);

procedure barbieri;
begin
  repeat
    wait (cliente);
    wait (mutex);
    coda--;
    signal (barbiere);
    signal (mutex);
    TAGLIO_CAPELLI;
  forever
end;

```

```

procedure clienti;
begin
  repeat
    wait (mutex);
    if coda < sedie {
      coda++;
      signal (mutex);
      signal (cliente);
      wait (barbiere);
      TAGLIO_CAPELLI;
    }
    else
      //vai via
      signal (mutex);
  forever
end;

```

Il barbiere si mette in attesa di un cliente.

Quando arriva un cliente, decrementa il valore della coda, cioè dei clienti in attesa. ovviamente effettua questa operazione bloccando il mutex.

Dopodichè segnala che è libero ed effettua il taglio.

Il cliente verifica se ci sono sedie dove mettersi in attesa.

Se non ce ne sono, va via.

Se ce ne sono, incrementa il valore della coda, cioè dei clienti in attesa. Questa operazione viene effettuata bloccando il mutex.

Dopodichè segnala la sua presenza e quindi si mette in attesa che il barbiere si liberi. Una volta libero, si fa tagliare i capelli.

7.12 Casi di studio: sincronizzazione dei thread POSIX

Nel caso di thread POSIX sono a disposizione i mutex per la mutua esclusione e le variabili di condizione per il controllo della sincronizzazione tra processi. Un mutex è in realtà un semaforo binario.

Visto che i thread POSIX possono essere implementati sia come thread kernel che come thread utente, allora i mutex possono essere implementati a livello kernel oppure utente (o anche mediante implementazione ibrida). Analogico discorso vale per le variabili di condizione.

CAPITOLO 8. Sincronizzazione dei processi: message passing

I processi scambiano informazioni attraverso la *comunicazione interprocesso* (IPC – InterProcess Communication). Questo capitolo illustra la semantica dello scambio di messaggi e il ruolo del SO nel bufferizzare e inoltrare i messaggi.

Lo scambio di messaggi (*message passing*) si adatta a diverse situazioni in cui lo scambio di informazioni tra processi gioca un ruolo fondamentale.

Uno dei suoi usi più importanti è nel modello *client-server*, in cui un processo *server* fornisce un servizio, e altri processi, chiamati *client*, gli inviano messaggi per utilizzare tale servizio.

Un altro importante utilizzo dello scambio di messaggi è nei protocolli di alto livello per lo scambio di email.

8.1 Panoramica sul message passing

Quando i processi interagiscono fra loro devono soddisfare due requisiti fondamentali: devono essere sincronizzati per garantire la mutua esclusione ed hanno bisogno di scambiarsi informazioni per cooperare tra loro. Un modo per soddisfare entrambe le necessità è lo scambio di messaggi.

Questi messaggi possono essere scambiati sia da processi in esecuzione sullo stesso computer che su computer differenti connessi tramite una rete.

Il message passing è solitamente implementato mediante le primitive *send* (invio) e *receive* (ricevo).

Per scambiarsi messaggi due processi devono essere in parte sincronizzati. Non è possibile che un processo riceva un messaggio prima che il mittente lo abbia inviato.

8.1.1 Send bloccanti e non-bloccanti

Le due primitive possono essere *bloccanti* o *non bloccanti*.

Consideriamo la primitiva *send*, ci sono due possibilità:

- il processo mittente si blocca finchè il ricevente non riceve il messaggio
- il processo mittente continua la sua esecuzione subito dopo aver inviato il messaggio

Analogamente, per quanto riguarda un processo che effettua una *receive*, può accadere che:

- il processo riceve il messaggio inviato dal mittente e prosegue la sua esecuzione
- se non ci sono messaggi in arrivo il processo si blocca in attesa di messaggi o continua l'esecuzione rinunciando a ricevere il messaggio

Possiamo sintetizzare queste situazioni distinguendo tra *send* e *receive* *bloccanti* e *non bloccanti*:

- una *send bloccante* blocca il processo mittente finchè il messaggio da inviare non viene consegnato al processo destinatario; questa metodologia di message passing prende il nome di *scambio di messaggi sincrono*
- una *send non bloccante* consente a un mittente di proseguire la propria esecuzione dopo aver effettuato una chiamata *send*, senza preoccuparsi dell'immediata consegna del messaggio; questa metodologia prende il nome di *scambio di messaggi asincrono*

In entrambe le metodologie, la *send* è tipicamente una *send non bloccante*.

8.1.2 Message passing sincrono e asincrono

Il *message passing sincrono* fornisce alcune proprietà per i processi utente e semplifica le azioni del kernel. Un processo mittente ha la garanzia che il messaggio inviato venga consegnato prima di poter continuare la

propria esecuzione. Questa caratteristica semplifica la progettazione dei processi concorrenti. Il kernel consegna il messaggio immediatamente se il processo destinatario ha già effettuato una chiamata *receive* per ricevere un messaggio; altrimenti, blocca il processo mittente finché il ricevente non effettua una *receive*. Tuttavia, l'uso delle *send bloccanti* presenta una controindicazione, può cioè ritardare un processo mittente in alcune situazioni.

Il *message passing asincrono* migliora la concorrenza tra i processi mittente e destinatario consentendo al processo mittente di continuare la propria esecuzione. Per realizzare questa metodologia, il kernel esegue il *buffering del messaggio*: quando un processo effettua una *send*, il kernel alloca un buffer nell'area di sistema e copia il messaggio nel buffer. In questo modo, il processo mittente è libero di accedere all'area di memoria che conteneva il testo del messaggio. Tuttavia, questa organizzazione presenta due svantaggi:

- spreco di memoria; coinvolge un sostanziale impiego di memoria per i buffer quando molti messaggi sono in attesa di essere consegnati
- spreco di CPU; un messaggio deve essere copiato due volte: una volta nel buffer di sistema quando viene effettuata la *send* e, successivamente, nell'area di messaggio del destinatario al momento della consegna.

8.1.3 Denominazione diretta e indiretta

Una problematica molto importante nello scambio di messaggi è l'*identificazione per nome (Naming) dei processi*, cioè la denominazione dei processi mittente e destinatario nelle chiamate *send* e *receive*.

I nomi dei processi mittente e destinatario, o sono indicati esplicitamente nelle istruzioni *send* e *receive*, o sono dedotti dal kernel in un altro modo.

Abbiamo due possibili schemi per specificare i processi nelle *send* e nelle *receive*.

Con l'*indirizzamento diretto* i processi mittente e destinatario dichiarano il proprio nome.

Tale tecnica può essere utilizzata in due modi.

Nella tecnica basata sui *nomi simmetrici* sia il mittente che il destinatario specificano i rispettivi nomi; in questo modo, un processo può decidere da quale processo ricevere un messaggio. Tuttavia deve conoscere il nome di ogni processo che vuole inviargli messaggi; cosa difficoltosa quando, ad esempio, i processi di applicazioni differenti vogliono comunicare.

Nella tecnica basata sui *nomi asimmetrici* il destinatario non fornisce il nome del processo da cui vuole ricevere un messaggio; il kernel inoltra un messaggio inviatogli da *qualche* processo.

Con l'*indirizzamento indiretto* i processi non specificano i rispettivi nomi nelle istruzioni *send* e *receive*.

In questo caso i messaggi non viaggiano direttamente dal mittente al destinatario, ma sono mandati ad una struttura dati condivisa che si compone di code che contengono contemporaneamente i messaggi. Questa struttura prende il nome di **mailbox** e possiede tre caratteristiche:

1. Ha un nome unico
2. Il proprietario della mailbox è tipicamente il processo che l'ha creata. Solo il proprietario del processo può ricevere i messaggi da una mailbox.
3. Ciascun processo che è a conoscenza del nome della mailbox gli può inviare messaggi (al processo *utente* della mailbox). In questo modo, i processi mittenti e destinatari utilizzando il nome di una mailbox, piuttosto che i rispettivi nomi, nelle istruzioni *send* e *receive*

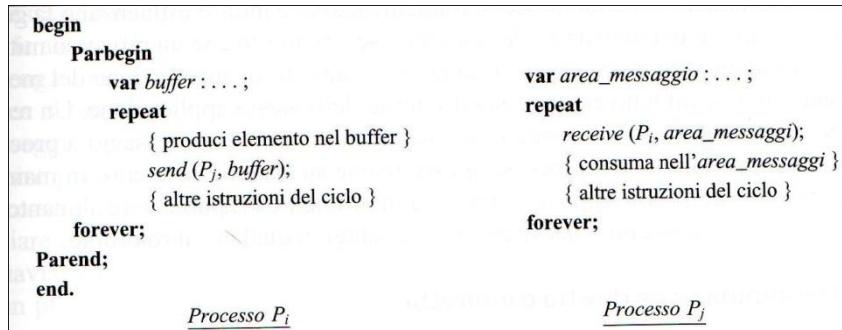
Le relazioni tra mittenti e destinatari possono essere *uno a uno*, *uno a molti*, *molti a molti*. L'associazione dei processi con le mailbox può essere *statica* o *dinamica*. Spesso una mailbox è associata staticamente ad un solo processo in particolare se la relazione tra mittente e destinatario è di tipo *uno a uno*. Mentre se la relazione è di tipo *uno a molti* o *molti a molti* allora si utilizzano mailbox con associazioni dinamiche.

VANTAGGI DI UNA MAILBOX

L'uso di una mailbox ha i seguenti vantaggi:

- *anonimato del destinatario*, infatti una mailbox consente al mittente di non conoscere l'identità del destinatario; inoltre se il SO consente di cambiare la proprietà di una mailbox dinamicamente, un processo può prontamente rilevare un altro servizio;
- *classificazione dei messaggi*, un processo può creare diverse mailbox e usare ognuna per ricevere i messaggi di un tipo specifico consentendo una semplice classificazione dei messaggi.

8.2 Produttore-consumatore mediante scambio di messaggi



La figura mostra una possibile soluzione del problema produttore-consumatore con buffer singolo, un solo produttore e un solo consumatore.

La soluzione non usa alcuna variabile condivisa. Il processo P_i , ovvero il processo produttore, ha una variabile chiamata *buffer*, mentre il processo P_j , il processo consumatore, ha una variabile chiamata *area_messaggio*.

Il processo produttore produce, inserisce nel *buffer* e invia il contenuto del *buffer* in un messaggio al consumatore. Il consumatore riceve il messaggio in *area_messaggio* e da lì lo consuma. La chiamata di sistema *send* blocca il processo produttore finché il messaggio non viene consegnato al consumatore, e la chiamata di sistema *receive* blocca il consumatore finché il messaggio non gli viene inviato.

Questa soluzione, come si può vedere, risulta molto più semplice rispetto alle soluzioni proposte nel capitolo precedente, tuttavia è restrittiva in quanto consente di avere un unico processo produttore ed un unico processo consumatore. Nel caso generale, per implementare più produttori e consumatori, occorre comunque utilizzare le tecniche di sincronizzazione dei processi viste precedentemente.

8.3 Casi di studio

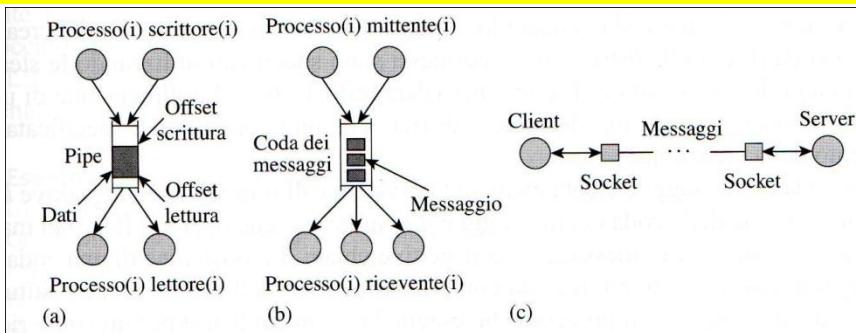
8.3.1 Message passing in Unix

Unix supporta tre funzioni di comunicazione tra processi denominate *pipe*, coda dei messaggi (*message queue*) e *socket*. Una pipe è una funzione per il trasferimento dei dati, mentre le code di messaggi e i socket sono usati per lo scambio di messaggi. Queste funzioni hanno una caratteristica in comune, cioè i processi possono comunicare senza conoscere le identità degli altri. Esse differiscono in dipendenza dell'applicazione.

Le *pipe anonymous* possono essere usate solo da processi che appartengono allo stesso albero di processi, mentre le *pipe con nome* possono essere usate anche da altri processi.

Le *code di messaggi* possono essere usate solo dai processi presenti nell' "Unix system domain", ovver nel dominio di Unix in esecuzione sul computer.

I *socket* possono essere usati dai processi nel dominio Unix e da quelli presenti in alcuni domini Internet.



PIPE

Una pipe è un meccanismo FIFO per il trasferimento dei dati tra processi lettori e scrittori.

Unix fornisce due tipi di pipe: *pipe con nome* e *pipe anonima*. Entrambi i tipi di pipe sono creati mediante la system call *pipe*. La differenza tra le due è la seguente:

- una *pipe con nome* corrisponde ad un elemento all'interno di una directory e può quindi essere utilizzata da qualsiasi processo mediante la system call *open*. Questo tipo di pipe viene mantenuta nel sistema finché non è rimossa attraverso la system call *unlink*;
- una *pipe anonima* non corrisponde ad alcun elemento all'interno di una directory; può essere usata solo dal suo creatore e dai suoi discendenti nell'albero dei processi. Il kernel cancella una pipe anonima quando non esistono più lettori o scrittori.

Il kernel tratta una pipe come una coda mantenendo due offset: un offset è usato per scrivere i dati nella pipe e l'altro per leggere i dati dalla pipe. Questa organizzazione fa in modo che un processo non possa modificare gli offset in alcun modo, se non mediante la lettura o scrittura dei dati.

I dati scritti vengono inseriti nella pipe utilizzando l'offset di scrittura che viene incrementato del numero di byte scritti. I dati scritti da più scrittori sono mischiati se vengono scritti in maniera interfogliata. Se una pipe è piena, un processo che vuole scrivere dati al suo interno viene posto in sleep.

I dati vengono letti utilizzando l'offset di lettura che viene incrementato del numero di byte letti. Un processo che legge dati da una pipe vuota viene posto in sleep.

CODA DEI MESSAGGI

Una coda di messaggi in Unix è analoga a una mailbox. Viene creata e posseduta da un processo. Altri processi possono inviare o ricevere messaggi verso oppure da una coda. Le dimensioni della coda sono specificate al momento della creazione.

Una coda di messaggi è creata mediante la system call *msgget (key, flag)* dove *key* specifica il nome della coda dei messaggi e *flag* indica alcune opzioni.

Se un processo effettua una chiamata *msgget* con una chiave che corrisponde al nome di una coda di messaggi esistente, il kernel restituisce l'identificativo della coda. In questo modo una coda può essere usata da qualsiasi processo nel sistema.

Se la chiave in una chiamata *msgget* non corrisponde al nome di una coda di messaggi esistente, il kernel crea una nuova coda, imposta la chiave come suo nome e restituisce l'identificativo della coda. Il processo che effettua la chiamata diventa il proprietario della coda.

SOCKET

Una socket è semplicemente un terminale di un canale di comunicazione. Possono essere utilizzate per la comunicazione tra processi all'interno del dominio Unix e nel dominio Internet.

Un canale di comunicazione tra un client e un server viene impostato come segue: i processi client e server creano ciascuno una socket. Queste due socket sono connesse per creare un canale di comunicazione al fine

di inviare e ricevere messaggi. Il server può impostare canali di comunicazione con più client simultaneamente.

Il problema dei nomi è affrontato come segue: il server collega la sua socket a un indirizzo valido nel dominio in cui il socket sarà utilizzato. L'indirizzo a questo punto viene reso noto nel dominio. Un processo client utilizza l'indirizzo per eseguire una *connect* tra la propria socket e quella del server. Questo metodo evita l'uso degli identificativi dei processi nella comunicazione.

Un server crea una socket mediante la chiamata di sistema:

$$s = \text{socket}(\text{domain}, \text{type}, \text{protocol})$$

La chiamata *socket* restituisce un identificatore di socket al processo.

Un client crea una socket mediante la chiamata *socket* e tenta di connetterlo al socket di un server utilizzando la system call:

$$\text{connect}(cs, \text{server_socket_addr}, \text{server_socket_addrlen})$$

La comunicazione tra un client e un server è implementata mediante le chiamate *read* e *write* o *send* e *receive*.

CAPITOLO 9. Deadlock

Un **deadlock** è una situazione in cui un insieme di processi attende indefinitamente degli eventi (che potrebbero non verificarsi mai), ciascuno dei quali può essere generato solo da altri processi dell'insieme. Un deadlock pregiudica il servizio utente, il throughput e l'efficienza delle risorse.

I SO usano diversi approcci per la gestione dei deadlock. In questo capitolo si discuteranno le tecniche adottate dai SO per gestione dei deadlock.

9.1 Definizione di deadlock

Quando due o più processi interagiscono, a volte possono mettersi in una situazione di stallo dalla quale non possono uscire; questa situazione è chiamata deadlock. Un deadlock si verifica quando due o più processi si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa. Ogni processo è dunque in attesa di un evento che potrebbe non verificarsi mai.

In un SO si possono verificare vari tipi di deadlock, come:

- il *deadlock generato da risorse* dove un processo rimane in attesa di una risorsa "bloccata" da un altro processo;
- il *deadlock di sincronizzazione* dove un processo rimane in attesa di un'operazione di un altro processo che, però, non la effettua;
- il *deadlock di comunicazione* dove un processo rimane in attesa di un messaggio da parte di un altro processo che, però, non lo invia.

Solitamente un SO deve principalmente risolvere i deadlock di risorsa in quanto l'allocazione delle risorse è una responsabilità del SO. Le altre due forme di deadlock sono gestite raramente dal SO dato che ci si aspetta che i processi utente gestiscano tali deadlock.

9.2 I deadlock nell'allocazione delle risorse

Un SO gestisce numerose risorse di tipo differente.

I processi usano risorse, che possono essere sia hardware (quali la memoria, i dispositivi di I/O, ecc.) sia software (quali i file, i semafori, i monitor, ecc.). Un SO può avere a disposizione diverse risorse di una stessa tipologia, quindi usiamo il termine:

- *unità di risorsa* per identificare una risorsa di un certo tipo
- *classe di risorse* per indicare l'insieme di tutte le unità di risorsa di un certo tipo

Per la gestione delle risorse, il kernel gestisce una *tabella delle risorse* per tener traccia dello stato di allocazione di una risorsa. Questa tabella contiene varie informazioni, come il nome della risorsa, la tipologia della risorsa, l'indirizzo della risorsa e lo stato della risorsa.

Quando un processo richiede una risorsa, si verifica se essa è allocata o meno ad un altro processo:

- se risulta già allocata, allora il processo va nello stato *blocked*
- se risulta libera, gli viene allocata e il processo va nello stato *ready*

Per poter utilizzare una risorsa, è necessario la seguente sequenza di eventi:

- *richiesta* della risorsa
- *allocazione* (intesa come uso) della risorsa
- *rilascio* della risorsa

La tabella seguente descrive questi tre tipi di eventi.

Evento	Descrizione
Richiesta	Un processo richiede una risorsa tramite una chiamata di sistema. Se la risorsa è libera, il kernel la alloca al processo immediatamente; altrimenti, cambia lo stato del processo a <i>blocked</i> .
Allocazione	Il processo diventa <i>holder</i> della risorsa a esso allocata. Le informazioni sullo stato della risorsa vengono aggiornate e lo stato del processo diventa <i>ready</i> .
Rilascio	Un processo rilascia una risorsa tramite una chiamata di sistema. Se vari processi sono bloccati sull'evento allocazione della risorsa, il kernel usa alcune regole, come l'allocazione FCFS, per decidere a quale processo allocare la risorsa.

9.2.1 Condizioni per un deadlock di risorsa

Un deadlock di risorsa si verifica quando si verificano quattro condizioni contemporaneamente:

1. *Condizioni di mutua esclusione*

Almeno una delle risorse del sistema deve essere "non condivisibile" (ossia deve essere usata da un processo alla volta oppure essere libera)

2. *Condizione di possesso e attesa (hold and wait)*

Un processo continua a tenere la risorsa allocata (che potrebbe servire ad altri processi) ma al tempo stesso, esso stesso, è in attesa di altre risorse per poter completare le sue operazioni

3. *Condizioni di assenza di prelazione*

Una risorsa allocata a un processo non può essere rimossa in modo forzato da questo processo per poter essere assegnata ad un altro processo

4. *Condizione di attesa circolare*

Esiste nel sistema una catena circolare dei processi, ognuno dei quali aspetta il rilascio di una risorsa da parte del processo che lo segue; ad esempio il processo Pi aspetta Pj, Pj aspetta Pk e Pk aspetta Pi

9.2.2 Modelli dello stato di allocazione delle risorse

Per stabilire se un insieme di processi è in deadlock è necessario analizzare sia le informazioni relative alle risorse allocate sia quelle relative alla richieste di risorse in attesa. Queste informazioni costituiscono lo *stato di allocazione delle risorse* di un sistema.

Per rappresentare lo stato di allocazione di un sistema si usano due tipi di modelli. Un *modello basato su grafo* o un *modello basato su matrice*.

MODELLO BASATO SU GRAFO

Un grafo di richiesta (RRAG) contiene due tipi di *nodi*:

- i nodi processi rappresentati da cerchi
- i nodi risorsa rappresentati da rettangoli e rappresentano classi di risorse; il numero di pallini nei nodi risorsa indica quante unità di quella classe esistono nel sistema

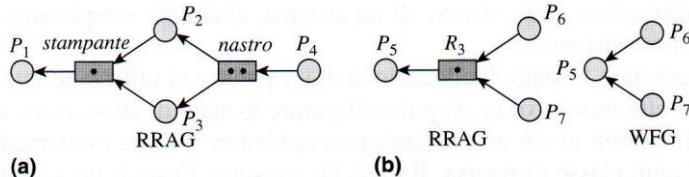
Per collegare i nodi processi e i nodi risorsa si usano degli *archi* che possono essere di due tipi:

- un *arco di allocazione* è diretto da un nodo risorsa a un nodo processo e indica che la risorsa è assegnata al processo
- un *arco di richiesta* è diretto da un nodo processo a un nodo risorsa e indica che il processo è attualmente bloccato in attesa della risorsa

Visto che ogni risorsa può avere più istanze (ad esempio, 5 stampanti, 2 processori, ecc), ciascuna di loro si rappresenta con un pallino all'interno del rettangolo. Occorre notare che un *arco di richiesta* è diretto soltanto

verso un rettangolo, mentre un *arco di assegnazione* deve partire da un'istanza della risorsa, cioè da uno dei pallini del rettangolo.

Un grafo di attesa (WFG) viene usato quando, in un sistema, una classe di risorsa contiene una sola unità di risorsa. Esso può rappresentare lo stato di allocazione in modo più conciso di un RRAG se ogni classe di risorsa nel sistema contiene solo un'unità di risorsa. Il WFG contiene i nodi solo di un tipo, detti, *nodi processo*. Un arco nel WFG rappresenta il fatto che il processo P_i è bloccato su una richiesta per una risorsa attualmente allocata al processo P_j (cioè il processo P_i sta attendendo che il processo P_j rilasci una risorsa).



La figura mostra un grafo RRAG e un grafo WFG equivalenti quando ogni classe di risorsa contiene soltanto una unità di risorsa. Come si può vedere il grafo RRAG richiede un arco in più rispetto al grafo WFG.

Se un grafo non contiene cicli, nessun processo del sistema subisce uno stallo; mentre se un grafo contiene un ciclo può verificarsi un deadlock (dipende dal numero di istanze della risorsa). In pratica, in un sistema dove ogni classe di risorsa contiene una singola unità di risorsa, la presenza di un ciclo implica un deadlock. Mentre, se una classe di risorsa contiene più di una unità di risorsa, la presenza di un ciclo non necessariamente implica l'esistenza di un deadlock.

MODELLO BASATO SU MATRICE

Nel modello basato su matrice, lo stato di allocazione di un sistema è rappresentato da due matrici:

- la *matrice delle risorse allocate* indica quante unità di risorsa di ogni classe di risorse sono allocate ad ogni processo del sistema
- la *matrice delle risorse richieste* tiene conto delle richieste in attesa e, in pratica, indica quante unità di risorsa di ogni classe di risorsa sono state richieste da ogni processo del sistema

Se un sistema contiene n processi ed r classi di risorsa, ognuna di queste matrici è una matrice $n \times r$.

In alcuni casi vengono utilizzate anche tabelle ausiliarie per rappresentare informazioni aggiuntive. Due di queste matrici sono la *matrice delle risorse totali* e la *matrice delle risorse disponibili*.

	Stampante	Nastro		Stampante	Nastro		Stampante	Nastro
P_i	0	1		P_i	1	0		
P_j	1	0		P_j	0	1		
P_k	0	1		P_k	0	0		
Risorse allocate						Risorse richieste		
							Risorse totali	1 2
							Risorse libere	0 0

9.3 Gestione dei deadlock

Per trattare adeguatamente le situazioni di stallo si possono impiegare tre diversi approcci.

Approccio	Descrizione
Rilevamento e risoluzione dei deadlock	Il kernel analizza lo stato della risorsa per controllare se esiste un deadlock. Se è così, interrompe alcuni processi e assegna le risorse allocate ad altri processi affinché il deadlock cessi di esistere.
Prevenzione dei deadlock	Il kernel usa una politica di allocazione delle risorse che assicura che le quattro condizioni per i deadlock di risorsa menzionate in Tabella 8.2 non si verifichino simultaneamente. Questo rende impossibili i deadlock.
Evitare i deadlock	Il kernel analizza lo stato di allocazione per determinare se l'accettazione di una richiesta di risorsa può determinare un deadlock. Vengono accettate solo richieste che non conducono a deadlock, le altre vengono tenute in attesa finché possono essere accettate. In questo modo, non si verificano deadlock.

Nell'approccio ***individuazione e risoluzione dei deadlock*** il kernel interrompe alcuni processi quando individua un deadlock. Questo permette di liberare le risorse assegnate al processo interrotto, che possono essere allocate, in tal modo, ad altri processi che ne hanno fatto richiesta. I processi interrotti vengono poi rieseguiti. Il costo di questo approccio comprende sia il costo dell'individuazione dei deadlock e sia il costo delle riesecuzione dei processi interrotti.

Nell'approccio ***prevenire i deadlock*** il kernel fa in modo che le quattro condizioni elencate in precedenza non si verifichino simultaneamente.

Nell'approccio *evitare i deadlock* il kernel accetta una richiesta di risorsa solo se constata che, accettando la richiesta, non si arriverà ad un deadlock successivamente; altrimenti, mantiene la richiesta in attesa finché può essere accettata. Ne consegue che un processo può andare incontro a lunghi ritardi per ottenere una risorsa.

9.4 Individuazione e risoluzione dei deadlock

Questo approccio non tenta di impedire il verificarsi dei deadlock, ma, al contrario, permette che si abbia il deadlock, per poi provare a scoprire quando ciò accade ed, infine, provare a risolvere la situazione di stallo dopo che si è verificata.

In pratica, il sistema deve fornire:

- un algoritmo che esamina lo stato del sistema per stabilire se si è verificato uno stallo
 - una tecnica per permettere al sistema di ricominciare a funzionare

Se ogni classe di risorse nel sistema contiene una sola unità di risorsa, questo controllo può essere fatto verificando la presenza di un ciclo in un grafo. Tuttavia, se le classi di risorse possono contenere più di una unità di risorse devono essere utilizzati algoritmi molto complessi basati sui grafici. Di queste soluzioni ne discuteremo nel paragrafo 9.5.

9.4.1 Individuazione dei deadlock

Discussiamo invece di un approccio che individua e risolve i deadlock basandosi sulle matrici.

Per determinare se c'è un deadlock oppure no nel sistema, occorre provare a costruire sequenze di eventi in base al quale tutti i processi *blocked* possono avere le risorse che hanno richiesto. Se si riesce con successo a costruire tale sequenza allora nel sistema non c'è presenza di deadlock; altrimenti ci sono processi in condizione di stallo.

Esempio 8.6 – Individuazione dei deadlock

Esempio 8.6 – Individuazione dei deadlock
 Si consideri il seguente stato di allocazione di un sistema con 10 unità di una classe di risorse R_1 e tre processi P_1 , P_2 e P_3 :

R1	R1	R1
P_1 4	P_1 6	Risorse totali 10
P_2 4	P_2 2	Risorse libere 0
P_3 2	P_3 0	
Risorse allocate	Risorse richieste	

Il processo P_3 è in stato *running* perché non è bloccato da una richiesta di risorse. Tutti i processi nel sistema possono terminare nel seguente modo: il processo P_3 termina e rilascia 2 unità della risorsa a esso allocata. Queste unità possono essere allocate a P_2 . Quando questo termina, 6 unità della risorsa possono essere allocate a P_1 . Così, non c'è alcun processo *bloccato* quando termina la simulazione, quindi non c'è deadlock nel sistema.

Se le richieste dai processi P_1 e P_2 fossero per 6 e 3 unità, rispettivamente, nessuno di loro potrebbe terminare anche dopo che il processo P_3 ha rilasciato 2 unità di risorsa. Questi processi sarebbero in stato *blocked* al termine della simulazione, e quindi sarebbero in deadlock allo stato attuale del sistema.

In questo esempio è stato assunto che:

- un processo *running* termini la sua esecuzione senza effettuare ulteriori richieste di risorse
- non è stato seguito un ordine preciso con il quale i processi *blocked* diventano *running* o con il quale i processi *running* terminano
- anche se un sistema non ha deadlock nello stato attuale, potrebbe comunque averne successivamente; per questo motivo bisogna eseguire ripetutamente l'individuazione dei deadlock durante il funzionamento del SO

9.4.2 Risoluzione dei deadlock

Supponiamo che siano stati individuati dei deadlock nel sistema. A questo punto è necessaria qualche tecnica che permetta al sistema di ricominciare a funzionare. Una delle tecniche più utilizzate è la **tecnica della risoluzione del deadlock mediante prelazione**. In pratica, se vi sono più processi in stato di deadlock, la *risoluzione dei deadlock mediante prelazione* comporta l'interruzione dei deadlock per assicurare l'avanzamento di alcuni di questi processi. Ciò può essere realizzato interrompendo uno o più processi e allocando le loro risorse a qualche altro processo in modo tale che possa completare le sue operazioni. Ogni processo interrotto è detto *vittima* della risoluzione del deadlock.

La scelta delle *vittime* viene fatta tramite criteri quali la priorità dei processi, le risorse già utilizzate, ecc.

9.5 Caratterizzazione dei deadlock delle risorse tramite i modelli a grafi

La *caratterizzazione del deadlock* è una dichiarazione delle caratteristiche essenziali di un deadlock.

Nel paragrafo precedente abbiamo presentato un algoritmo di individuazione dei deadlock che adotta il modello basato su matrice e secondo questo algoritmo possiamo caratterizzare un deadlock come una situazione in cui non è possibile costruire una sequenza di eventi che porta al completamento di tutti i processi del sistema.

In questo paragrafo discutiamo la caratterizzazione dei deadlock usando i modelli basati su grafo. Tra le condizioni necessarie per un deadlock c'è anche l'attesa circolare, che si manifesta in un ciclo nei modelli basati sui grafi. Come già detto, un ciclo è una condizione sufficiente per un deadlock in alcuni sistemi (se una classe di risorsa contiene una sola unità di risorsa) ma non in altri (se una classe di risorsa contiene più di una unità di risorsa).

9.5.1 Sistemi Singola-Istanza

In un sistema a singola istanza, ogni classe di risorsa contiene una sola istanza di risorsa.

L'esistenza di un ciclo in un grafo implica una relazione di attesa reciproca per un insieme di processi. Un ciclo è, quindi, una condizione necessaria e sufficiente per concludere che esiste un deadlock nel sistema.

9.5.2 Sistemi Istanza-Multipla

Un ciclo non è una condizione sufficiente per un deadlock nei sistemi a istanza multipla perché le classi di risorsa possono contenere diverse unità di risorsa. Affinché un processo sia in deadlock è necessario che tutti i processi che posseggono unità di una risorsa da esso richiesta, siano anch'essi in deadlock.

In pratica, la presenza di un taglio in un RRAG è una condizione necessaria e sufficiente per l'esistenza di un deadlock in un sistema a multipla istanza.

9.6 Prevenzione dei deadlock

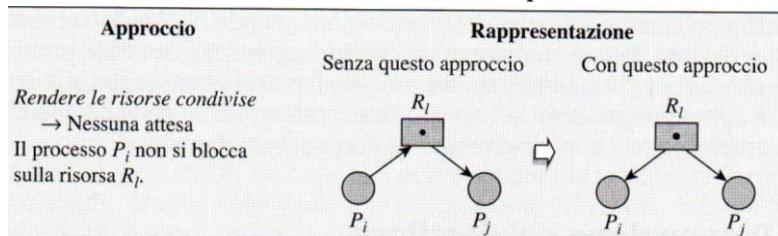
In questo approccio si cerca di evitare i deadlock annullando una o più *condizioni necessarie*.

Condizione	Approccio
Mutua esclusione	Spool di tutto
Hold and wait	Richiede inizialmente tutte le risorse
No prerilascio	Portare via le risorse
Attesa circolare	Ordinare numericamente le risorse

In pratica, il kernel può usare una politica di allocazione delle risorse che assicuri che un di queste quattro condizioni non possa verificarsi.

MUTUA ESCLUSIONE

In un sistema non esisterebbero deadlock se tutte le risorse potessero essere rese condivisibili. Infatti, in questo modo, in un RRAG ci sarebbero solo archi di allocazione, quindi non si avrebbero mai attese circolari.



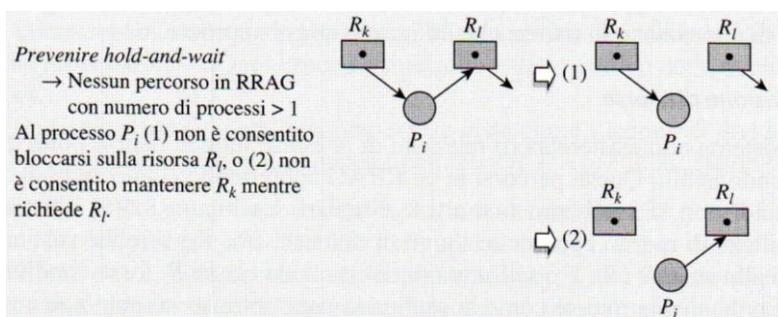
Un problema è che alcune risorse sono non condivisibili, ma si può superare tale problema creando dispositivi virtuali, come ad esempio una stampante virtuale da allocare a vari processi.

Un altro problema è che questo approccio non può funzionare per risorse software come file condivisi, che dovrebbero essere modificati in maniera mutuamente esclusiva per evitare race condition.

HOLD AND WAIT

Per evitare la condizione relativa al possesso e attesa, è necessario che un processo che abbia acquisito delle risorse non possa effettuare richieste di risorse, oppure che un processo bloccato su una richiesta di risorsa non possa poter impegnare altre risorse.

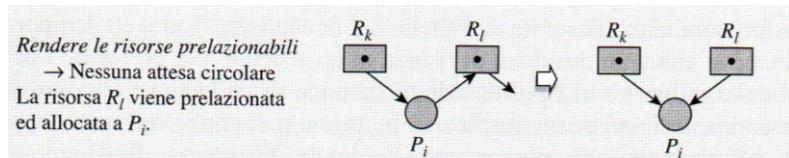
In un RRAG, i percorsi che coinvolgono più di un processo non possono comparire, quindi non possono esistere percorsi circolari.



Una semplice politica per l'implementazione di questo approccio è consentire a un processo di effettuare solo una richiesta di risorse durante la propria esecuzione in cui chiede tutte le risorse di cui necessita.

NO PRELAZIONE

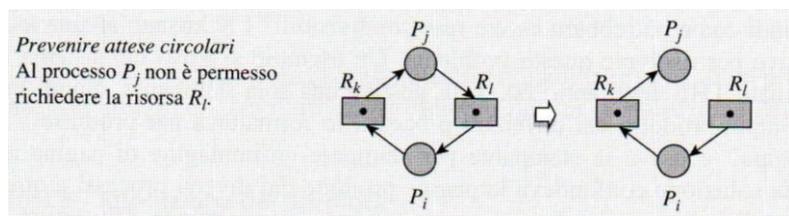
Se le risorse sono rese prelazionabili, il kernel può assicurare che i processi abbiano tutte le risorse di cui necessitano, il che evita percorsi circolari nei RRAG.



L'unico problema è che i dispositivi sequenziali di I/O non possono essere prelazionati.

ATTESA CIRCOLARE

Un'attesa circolare può dipendere dalla condizione hold-and-wait, che a sua volta è conseguenza delle condizioni di mutua esclusione e assenza di prelazione; quindi, non si verifica se non si verifica nessuna di queste condizioni. Le attese circolari possono essere evitate separatamente non consentendo ad alcuni processi di attendere determinate risorse.



9.7 Evitare i deadlock

La politica di evitare i deadlock ammette una richiesta di risorsa solo se si riesce a stabilire che l'accettazione della richiesta non può condurre a un deadlock né immediatamente né successivamente. In caso contrario, la richiesta è messa in attesa finché non viene ammessa.

Questa soluzione è possibile solo se il sistema è capace di mantenere delle informazioni sulle risorse disponibili al sistema e sulle risorse che ogni processo può potenzialmente richiedere.

Ma come può il kernel determinare se può verificarsi un deadlock successivamente? Si utilizza un approccio conservativo di questo genere: ogni processo dichiara il massimo numero di unità di risorsa di ogni classe che può richiedere. Il kernel permette che un processo richieda queste unità di risorsa a stadi successivi, ossia, poche unità di risorsa alla volta, in base al massimo numero dichiarato dal processo, e utilizza una tecnica di analisi nel caso peggiore per controllare la possibilità di deadlock successivi. Questo approccio è conservativo in quanto un processo può terminare il suo funzionamento senza richiedere il massimo numero dichiarato di unità.

Uno degli algoritmi che adotta tale approccio è l'*algoritmo del banchiere*. Tuttavia, per la maggior parte dei sistemi è impossibile conoscere in anticipo le risorse che richiederà un processo, per cui è spesso impossibile evitare del tutto i deadlock.

9.7.1 Algoritmo del banchiere

L'*algoritmo del banchiere* è un algoritmo di scheduling utilizzato per prevenire i deadlock nell'allocazione delle risorse. In particolare questo algoritmo può indicare se un sistema (in particolare un SO) si venga a trovare in uno stato sicuro o meno nel caso assegnasse una risorsa ad uno dei processi richiedenti.

Ricordiamo che se il sistema è in uno *stato sicuro* allora può garantire che tutti i processi avranno termine, mentre se il sistema è in uno *stato non sicuro* allora non c'è nessuna garanzia di questo tipo.

E' chiamato *algoritmo del banchiere* perché i banchieri hanno bisogno di un algoritmo simile quando ammettono prestiti che collettivamente superano i fondi bancari e rilasciano ogni prestito a rate.

La tabella descrive le strutture dati di tipo array usate per memorizzare le seguenti informazioni:

Notazione	Significato
$Risorse_richieste_{j,k}$	Numero di unità della classe di risorsa R_k attualmente richieste dal processo P_j
$Risorse_massime_{j,k}$	Massimo numero di unità della classe di risorsa R_k di cui può aver bisogno il processo P_j
$Risorse_allocate_{j,k}$	Numero di unità della classe di risorsa R_k allocate al processo P_j
$Risorse_totali_allocate_k$	Numero totale di unità allocate della classe di risorsa R_k , ossia $\sum_j Risorse_allocate_{j,k}$
$Risorse_totali_k$	Numero totale di unità della classe di risorsa R_k appartenenti al sistema

Il kernel ammette il processo P_j solo se $Risorse_massime < Risorse_totali$.

FUNZIONAMENTO:

Le tecniche per evitare i deadlock sono implementate trasferendo il sistema da uno stato sicuro a un altro stato sicuro come descritto di seguito:

1. quando un processo effettua una richiesta, si calcola il nuovo stato in cui si troverebbe il sistema se la richiesta fosse ammessa. Chiameremo questo stato, *stato proiettato*;
2. se lo stato proiettato è uno stato sicuro, si ammette la richiesta aggiornando le matrici $Risorse_allocate$ e $Risorse_totali$; altrimenti, si tiene la richiesta in attesa;
3. quando un processo rilascia una o più risorse o termina la propria esecuzione, si esaminano tutte le richieste in attesa e si allocano quelle che porterebbero il sistema in un nuovo stato sicuro.

Tutto ciò può essere realizzato usando un algoritmo di *individuazione dei deadlock*, però apportando una modifica: il completamento di un processo P , sia *running* che *blocked*, può richiedere ($Risorse_massime - Risorse_allocate$) ulteriori unità di risorse di ogni classe di risorsa. Quindi l'algoritmo controlla che:

per ogni classe di risorsa: $Risorse_totali - Risorse_totali_allocate \geq Risorse_massime(P) - Risorse_allocate(P)$

Se questa condizione è soddisfatta, viene simulato il completamento del processo P . E poi, viene controllato se qualche altro processo può soddisfare questa condizione e così via.

ALGORITMO:**Algoritmo 8.2 Algoritmo del banchiere****Input**

<i>n</i>	: numero di processi;
<i>r</i>	: numero di classi di risorse;
<i>Blocked</i>	: insieme di processi;
<i>Running</i>	: insieme di processi;
<i>P_{processo_richiedente}</i>	: Processo che effettua la nuova richiesta di risorsa;
<i>Risorse_massime</i>	: array [1.. <i>n</i> , 1.. <i>r</i>] di integer;
<i>Risorse_allocate</i>	: array [1.. <i>n</i> , 1.. <i>r</i>] di integer;
<i>Risorse_richieste</i>	: array [1.. <i>n</i> , 1.. <i>r</i>] di integer;
<i>Risorse_totali_allocate</i>	: array [1.. <i>r</i>] di integer;
<i>Risorse_totali</i>	: array [1.. <i>r</i>] di integer;

Strutture dati

<i>Active</i>	: insieme di processi;
<i>fattibile</i>	: boolean;
<i>Nuova_richiesta</i>	: array [1.. <i>r</i>] di integer;
<i>Allocazione_simulata</i>	: array [1.. <i>n</i> , 1.. <i>r</i>] di integer;
<i>Risorse_totali_allocate_simulate</i>	: array [1.. <i>r</i>] di integer;

1. $Active = Running \cup Blocked;$
for $k = 1..r$
 $Nuova_richiesta[k] = Risorse_richieste[processo_richiedente, k];$
2. $Allocazione_simulata := Risorse_allocate;$
for $k = 1..r$ /* Calcolare lo stato di allocazione proiettato*/
 $Allocazione_simulata[processo_richiedente, k] :=$
 $Allocazione_simulata[processo_richiedente, k] + Nuova_richiesta[k];$
 $Risorse_totali_allocate_simulate[k] := Risorse_totali_allocate[k] + Nuova_richiesta[k];$
3. $fattibile = true;$
for $k = 1..r$ /* Controllare se lo stato di allocazione proiettato è fattibile */
if $Risorse_totali[k] < Risorse_totali_allocate_simulate[k]$ **then** $fattibile = false;$
4. **if** $fattibile = true$
then /* Controllare se lo stato di allocazione proiettato è uno stato di allocazione sicuro */
while l'insieme Active contiene un processo P_i tale che
Per ogni k , $Risorse_totali[k] - Risorse_totali_allocate_simulate[k]$
 $\geq Risorse_massime[i, k] - Allocazione_simulata[i, k]$
Rimuovi P_i da Active;
for $k = 1..r$
 $Risorse_totali_allocate_simulate[k] :=$
 $Risorse_totali_allocate_simulate[k] - Allocazione_simulata[i, k];$
5. **if** l'insieme Active è vuoto
then /* Lo stato di allocazione proiettato è uno stato di allocazione sicuro */
for $k = 1..r$ /* Cancellare la lista dalle richieste in attesa */
 $Risorse_richieste[processo_richiedente, k] := 0;$
for $k = 1..r$ /* Accettare la richiesta */
 $Risorse_allocate[processo_richiedente, k] :=$
 $Risorse_allocate[processo_richiedente, k] + Nuova_richiesta[k];$
 $Risorse_totali_allocate[k] := Risorse_totali_allocate[k] + Nuova_richiesta[k];$

Quando un processo effettua una nuova richiesta, viene inserita nella matrice *Risorse_richieste*, che memorizza le richieste in attesa di tutti i processi e viene invocato l'algoritmo con l'identificativo del processo richiedente. Quando un processo rilascia alcune risorse allocate per sé oppure termina le proprie operazioni, viene invocato l'algoritmo per ogni processo, la cui richiesta è in attesa.

L'algoritmo può essere descritto come segue: dopo alcune inizializzazioni al passo1, l'algoritmo simula l'accettazione della richiesta al passo2 calcolando lo stato proiettato. Il passo3 controlla se lo stato proiettato è fattibile, ossia se esistono sufficienti risorse libere per permettere l'accettazione della richiesta.

Per controllare se lo stato proiettato è uno stato sicuro, si controlla se il massimo numero di risorse di cui necessita ogni processo attivo (*running* o *blocked*) può essere soddisfatto allocando alcune delle risorse libere. In caso affermativo, l'algoritmo simula il suo completamento cancellandolo dall'insieme dei processi *active* e

rilasciando le risorse allocate per esso. Questa azione viene ripetuta finché non è possibile cancellare più processi dall'insieme *active*. Se al termine di questo passo l'insieme *active* è vuoto, lo stato proiettato è uno stato sicuro, quindi l'algoritmo cancella la richiesta dalla lista delle richieste in attesa e alloca le risorse richieste. Questa azione non sarà eseguita se lo stato proiettato non è né fattibile né sicuro, per cui la richiesta rimane in attesa.

La complessità computazionale di questo algoritmo è $O(n^2m)$ dove n è il numero di processi ed m è il numero di tipi di risorse (per ogni tipo possono essere disponibili più risorse).

9.8 Starvation

Un problema strettamente in relazione con i deadlock è la *starvation* per cui si intende l'impossibilità, da parte di un processo pronto all'esecuzione, di ottenere le risorse di cui necessita.

Un esempio tipico è il non riuscire a ottenere il controllo della CPU da parte di processi con priorità molto bassa, qualora vengano usati algoritmi di scheduling a priorità. Può capitare, infatti, che venga continuamente sottomesso al sistema un processo con priorità più alta.

Per evitare questi problemi si possono utilizzare, oltre ad algoritmi di scheduling diversi, come RR, le cosiddette tecniche di invecchiamento (aging).

Ovvero si provvede ad aumentare progressivamente, ad intervalli regolari, la priorità dei processi qualora questi non siano riusciti ad ottenere le risorse richieste. Questa fa sì che anche un processo con la priorità più bassa possa potenzialmente assumere quella più alta, riuscendo così ad ottenere, in un tempo massimo predefinito, quanto di cui necessita.

Altri usi del termine starvation sono in relazione alle risorse di accesso alla memoria o CPU: si dice che un programma è bandwith-starved quando la sua esecuzione è rallentata da un'insufficiente velocità di accesso alla memoria, o CPU-starved quando il processore è troppo lento per eseguire efficacemente il codice.

9.9 Casi di studio

Un SO gestisce numerose risorse di differenti tipi. L'overhead per l'individuazione e la risoluzione dei deadlock e delle tecniche per evitare i deadlock rende tali politiche di gestione dei deadlock poco interessanti nella pratica. Di conseguenza, un SO può utilizzare l'approccio di prevenzione dei deadlock. Inoltre, poiché la prevenzione dei deadlock vincola l'ordine in cui i processi richiedono le loro risorse, i SO tendono a gestire i deadlock separatamente per ogni tipo di risorsa come la memoria, i dispositivi di I/O, i file e le risorse del kernel.

9.9.1 Gestione dei deadlock in Unix

Unix, per la gestione dei deadlock fa affidamento sull'algoritmo dello struzzo. Questo approccio è adottato dalla maggior parte dei SO. Consiste semplicemente nell'ignorare il problema, e si basa sull'ipotesi che la maggior parte degli utenti preferisca una situazione occasionale di stallo ad una regola che imponga ad ogni utente di usare un unico processo, di aprire un unico file, o di usare un'unica copia di qualsiasi altra risorsa.